```
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(name_index);
    }


}
```

A16. **CONSTANT_Double_info.java**

Convience class for representing CONSTANT_Double_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Double subtype of a constant pool entry. */
public final class CONSTANT_Double_info extends cp_info{

    /** The actual value. */
    public double bytes;

    public CONSTANT_Double_info(double d){
        tag = 6;
        bytes = d;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Double_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 6)
            throw new ClassFormatError();
        bytes = in.readDouble();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeDouble(bytes);
        long l = Double.doubleToLongBits(bytes);
    }
}
```

A17. **CONSTANT_Fieldref_info.java**

Convience class for representing CONSTANT_Fieldref_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Fieldref subtype of a constant pool entry.  */
public final class CONSTANT_Fieldref_info extends cp_info{

    /** The index to the class that this field is referencing to. */
    public int class_index;

    /** The name and type index this field if referencing to. */
    public int name_and_type_index;

    /** Convenience constructor. */
    public CONSTANT_Fieldref_info(int class_index, int name_and_type_index) {
        tag = 9;
```

```
            this.class_index = class_index;
            this.name_and_type_index = name_and_type_index;
        }

        /** Used during input serialization by ClassFile only. */
        CONSTANT_Fieldref_info(ClassFile cf, DataInputStream in)
            throws IOException{
            super(cf, in);
            if (tag != 9)
                throw new ClassFormatError();
            class_index = in.readChar();
            name_and_type_index = in.readChar();
        }

        /** Used during output serialization by ClassFile only. */
        void serialize(DataOutputStream out)
            throws IOException{
            out.writeByte(tag);
            out.writeChar(class_index);
            out.writeChar(name_and_type_index);
        }

    }
```

A18. **CONSTANT_Float_info.java**

Convience class for representing CONSTANT_Float_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Float subtype of a constant pool entry. */
public final class CONSTANT_Float_info extends cp_info{

    /** The actual value. */
    public float bytes;

    public CONSTANT_Float_info(float f){
        tag = 4;
        bytes = f;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Float_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 4)
            throw new ClassFormatError();
        bytes = in.readFloat();
    }

    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(4);
        out.writeFloat(bytes);
    }

}
```

A19. **CONSTANT_Integer_info.java**

Convience class for representing CONSTANT_Integer_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;
```

```
/** Integer subtype of a constant pool entry.  */
public final class CONSTANT_Integer_info extends cp_info{

    /** The actual value. */
    public int bytes;

    public CONSTANT_Integer_info(int b) {
        tag = 3;
        bytes = b;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Integer_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 3)
            throw new ClassFormatError();
        bytes = in.readInt();
    }

    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeInt(bytes);
    }


}
```

A20. **CONSTANT_InterfaceMethodref_info.java**

Convience class for representing CONSTANT_InterfaceMethodref_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** InterfaceMethodref subtype of a constant pool entry.
 */
public final class CONSTANT_InterfaceMethodref_info extends cp_info{

    /** The index to the class that this field is referencing to. */
    public int class_index;

    /** The name and type index this field if referencing to. */
    public int name_and_type_index;

    public CONSTANT_InterfaceMethodref_info(int class_index,
                                            int name_and_type_index) {
        tag = 11;
        this.class_index = class_index;
        this.name_and_type_index = name_and_type_index;
    }


    /** Used during input serialization by ClassFile only. */
    CONSTANT_InterfaceMethodref_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 11)
            throw new ClassFormatError();
        class_index = in.readChar();
        name_and_type_index = in.readChar();
    }
```

```
    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(class_index);
        out.writeChar(name_and_type_index);
    }

}
```

A21. **CONSTANT_Long_info.java**

Convience class for representing CONSTANT_Long_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Long subtype of a constant pool entry.  */
public final class CONSTANT_Long_info extends cp_info{

    /** The actual value. */
    public long bytes;

    public CONSTANT_Long_info(long b){
        tag = 5;
        bytes = b;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Long_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 5)
            throw new ClassFormatError();
        bytes = in.readLong();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeLong(bytes);
    }

}
```

A22. **CONSTANT_Methodref_info.java**

Convience class for representing CONSTANT_Methodref_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Methodref subtype of a constant pool entry.
 */
public final class CONSTANT_Methodref_info extends cp_info{

    /** The index to the class that this field is referencing to. */
    public int class_index;

    /** The name and type index this field if referencing to. */
    public int name_and_type_index;
```

```
    public CONSTANT_Methodref_info(int class_index, int name_and_type_index) {
        tag = 10;
        this.class_index = class_index;
        this.name_and_type_index = name_and_type_index;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Methodref_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 10)
            throw new ClassFormatError();
        class_index = in.readChar();
        name_and_type_index = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(class_index);
        out.writeChar(name_and_type_index);
    }

}
```

A23. CONSTANT_NameAndType_info.java

Convience class for representing CONSTANT_NameAndType_info structures within ClassFiles.

```
import java.io.*;
import java.lang.*;

/** NameAndType subtype of a constant pool entry.
 */
public final class CONSTANT_NameAndType_info extends cp_info{

    /** The index to the Utf8 that contains the name. */
    public int name_index;

    /** The index fo the Utf8 that constains the signature. */
    public int descriptor_index;

    public CONSTANT_NameAndType_info(int name_index, int descriptor_index) {
        tag = 12;
        this.name_index = name_index;
        this.descriptor_index = descriptor_index;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_NameAndType_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 12)
            throw new ClassFormatError();
        name_index = in.readChar();
        descriptor_index = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
```

```
        out.writeChar(name_index);
        out.writeChar(descriptor_index);
    }

}
```

A24. **CONSTANT_String_info.java**

Convience class for representing CONSTANT_String_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** String subtype of a constant pool entry.
 */
public final class CONSTANT_String_info extends cp_info{

    /** The index to the actual value of the string. */
    public int string_index;

    public CONSTANT_String_info(int value) {
        tag = 8;
        string_index = value;
    }

    /** ONLY TO BE USED BY CLASSFILE! */
    public CONSTANT_String_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 8)
            throw new ClassFormatError();
        string_index = in.readChar();
    }

    /** Output serialization, ONLY TO BE USED BY CLASSFILE! */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(string_index);
    }

}
```

A25. **CONSTANT_Utf8_info.java**

Convience class for representing CONSTANT_Utf8_info structures within ClassFiles.

```
import java.io.*;
import java.lang.*;

/** Utf8 subtype of a constant pool entry.
 *  We internally represent the Utf8 info byte array
 *  as a String.
 */
public final class CONSTANT_Utf8_info extends cp_info{

    /** Length of the byte array. */
    public int length;

    /** The actual bytes, represented by a String. */
    public String bytes;

    /** This constructor should be used for the purpose
     *  of part creation. It does not set the parent
     *  ClassFile reference.
     */
```

```
    public CONSTANT_Utf8_info(String s) {
        tag = 1;
        length = s.length();
        bytes = s;
    }

    /** Used during input serialization by ClassFile only. */
    public CONSTANT_Utf8_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 1)
            throw new ClassFormatError();
        length = in.readChar();
        byte[] b = new byte[length];
        in.read(b, 0, length);

        // WARNING: String constructor is deprecated.
        bytes = new String(b, 0, length);
    }

    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(length);

        // WARNING: Handling of String coversion here might be problematic.
        out.writeBytes(bytes);
    }

    public String toString(){
        return bytes;
    }

}
```

A26. ConstantValue_attribute.java

Convience class for representing ConstantValue_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Attribute that allows for initialization of static variables in
 *  classes. This attribute will only reside in a field_info struct.
 */

public final class ConstantValue_attribute extends attribute_info{

    public int constantvalue_index;

    public ConstantValue_attribute(ClassFile cf, int ani, int al, int cvi){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        constantvalue_index = cvi;
    }

    public ConstantValue_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        constantvalue_index = in.readChar();
    }
```

```
    public void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2;
        super.serialize(out);
        out.writeChar(constantvalue_index);
    }

}
```

A27. **cp_info.java**

Convience class for representing cp_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Represents the common interface of all constant pool parts
 *  that all specific constant pool items must inherit from.
 *
 */
public abstract class cp_info{

    /** The type tag that signifies what kind of constant pool
     *  item it is */
    public int tag;

    /** Used for serialization of the object back into a bytestream. */
    abstract void serialize(DataOutputStream out) throws IOException;

    /** Default constructor. Simply does nothing. */
    public cp_info() {}

    /** Constructor simply takes in the ClassFile as a reference to
     *  it's parent
     */
    public cp_info(ClassFile cf) {}

    /** Used during input serialization by ClassFile only. */
    cp_info(ClassFile cf, DataInputStream in)
        throws IOException{
        tag = in.readUnsignedByte();
    }


}
```

A28. **Deprecated_attribute.java**

Convience class for representing Deprecated_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** A fix attributed that can be located either in the ClassFile,
 *  field_info or the method_info attribute. Mark deprecated to
 *  indicate that the method, class or field has been superceded.
 */
public final class Deprecated_attribute extends attribute_info{

    public Deprecated_attribute(ClassFile cf, int ani, int al){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
    }


    /** Used during input serialization by ClassFile only. */
```

```
    Deprecated_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
    }

}
```

A29. **Exceptions_attribute.java**

Convience class for representing Exceptions_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** This is the struct where the exceptions table are located.
 *  <br><br>
 *  This attribute can only appear once in a method_info struct.
 */
public final class Exceptions_attribute extends attribute_info{

    public int number_of_exceptions;
    public int[] exception_index_table;

    public Exceptions_attribute(ClassFile cf, int ani, int al, int noe,
                                int[] eit){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        number_of_exceptions = noe;
        exception_index_table = eit;
    }

    /** Used during input serialization by ClassFile only. */
    Exceptions_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        number_of_exceptions = in.readChar();
        exception_index_table = new int[number_of_exceptions];
        for (int i=0; i<number_of_exceptions; i++)
            exception_index_table[i] = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2 + (number_of_exceptions*2);
        super.serialize(out);
        out.writeChar(number_of_exceptions);
        for (int i=0; i<number_of_exceptions; i++)
            out.writeChar(exception_index_table[i]);
    }

}
```

A30. **field_info.java**

Convience class for representing field_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/**   Represents the field_info structure as specified in the JVM specification.
 */
public final class field_info{

    public int access_flags;
    public int name_index;
```

```
    public int descriptor_index;
    public int attributes_count;
    public attribute_info[] attributes;

    /** Convenience constructor. */
    public field_info(ClassFile cf, int flags, int ni, int di){
        access_flags = flags;
        name_index = ni;
        descriptor_index = di;
        attributes_count = 0;
        attributes = new attribute_info[0];
    }

    /** Constructor called only during the serialization process.
     *  <br><br>
     *  This is intentionally left as package protected as we
     *  should not normally call this constructor directly.
     *  <br><br>
     *  Warning: the handling of len is not correct (after String s =...)
     */
    field_info(ClassFile cf, DataInputStream in)
        throws IOException{
        access_flags = in.readChar();
        name_index = in.readChar();
        descriptor_index = in.readChar();
        attributes_count = in.readChar();
        attributes = new attribute_info[attributes_count];
        for (int i=0; i<attributes_count; i++){
            in.mark(2);
            String s = cf.constant_pool[in.readChar()].toString();
            in.reset();
            if (s.equals("ConstantValue"))
                attributes[i] = new ConstantValue_attribute(cf, in);
            else if (s.equals("Synthetic"))
                attributes[i] = new Synthetic_attribute(cf, in);
            else if (s.equals("Deprecated"))
                attributes[i] = new Deprecated_attribute(cf, in);
            else
                attributes[i] = new attribute_info.Unknown(cf, in);
        }
    }

    /** To serialize the contents into the output format.
     */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeChar(access_flags);
        out.writeChar(name_index);
        out.writeChar(descriptor_index);
        out.writeChar(attributes_count);
        for (int i=0; i<attributes_count; i++)
            attributes[i].serialize(out);
    }

}
```

A31. **InnerClasses_attribute.java**

Convience class for representing InnerClasses_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** A variable length structure that contains information about an
 *  inner class of this class.
 */
```

```
public final class InnerClasses_attribute extends attribute_info{

    public int number_of_classes;
    public classes[] classes;

    public final static class classes{
        int inner_class_info_index;
        int outer_class_info_index;
        int inner_name_index;
        int inner_class_access_flags;
    }

    public InnerClasses_attribute(ClassFile cf, int ani, int al,
                                  int noc, classes[] c){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        number_of_classes = noc;
        classes = c;
    }

    /** Used during input serialization by ClassFile only. */
    InnerClasses_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        number_of_classes = in.readChar();
        classes = new InnerClasses_attribute.classes[number_of_classes];
        for (int i=0; i<number_of_classes; i++){
            classes[i] = new classes();
            classes[i].inner_class_info_index = in.readChar();
            classes[i].outer_class_info_index = in.readChar();
            classes[i].inner_name_index = in.readChar();
            classes[i].inner_class_access_flags = in.readChar();
        }
    }

    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2 + (number_of_classes * 8);
        super.serialize(out);
        out.writeChar(number_of_classes);
        for (int i=0; i<number_of_classes; i++){
            out.writeChar(classes[i].inner_class_info_index);
            out.writeChar(classes[i].outer_class_info_index);
            out.writeChar(classes[i].inner_name_index);
            out.writeChar(classes[i].inner_class_access_flags);
        }
    }

}
```

A32. **LineNumberTable_attribute.java**

Convience class for representing LineNumberTable_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Determines which line of the binary code relates to the
 *  corresponding source code.
 */
public final class LineNumberTable_attribute extends attribute_info{
```

```
    public int line_number_table_length;
    public line_number_table[] line_number_table;

    public final static class line_number_table{
        int start_pc;
        int line_number;
    }

    public LineNumberTable_attribute(ClassFile cf, int ani, int al, int lntl,
                                     line_number_table[] lnt){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        line_number_table_length = lntl;
        line_number_table = lnt;
    }

    /** Used during input serialization by ClassFile only. */
    LineNumberTable_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        line_number_table_length = in.readChar();
        line_number_table = new
LineNumberTable_attribute.line_number_table[line_number_table_length];
        for (int i=0; i<line_number_table_length; i++){
            line_number_table[i] = new line_number_table();
            line_number_table[i].start_pc = in.readChar();
            line_number_table[i].line_number = in.readChar();
        }
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2 + (line_number_table_length * 4);
        super.serialize(out);
        out.writeChar(line_number_table_length);
        for (int i=0; i<line_number_table_length; i++){
            out.writeChar(line_number_table[i].start_pc);
            out.writeChar(line_number_table[i].line_number);
        }
    }

}
```

A33. LocalVariableTable_attribute.java

Convience class for representing LocalVariableTable_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Used by debugger to find out how the source file line number is linked
 *  to the binary code. It has many to one correspondence and is found in
 *  the Code_attribute.
 */
public final class LocalVariableTable_attribute extends attribute_info{

    public int local_variable_table_length;
    public local_variable_table[] local_variable_table;

    public final static class local_variable_table{
        int start_pc;
        int length;
```

```
        int name_index;
        int descriptor_index;
        int index;
    }

    public LocalVariableTable_attribute(ClassFile cf, int ani, int al,
                                        int lvtl, local_variable_table[] lvt){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        local_variable_table_length = lvtl;
        local_variable_table = lvt;
    }

    /** Used during input serialization by ClassFile only. */
    LocalVariableTable_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        local_variable_table_length = in.readChar();
        local_variable_table = new
LocalVariableTable_attribute.local_variable_table[local_variable_table_length];
        for (int i=0; i<local_variable_table_length; i++){
            local_variable_table[i] = new local_variable_table();
            local_variable_table[i].start_pc = in.readChar();
            local_variable_table[i].length = in.readChar();
            local_variable_table[i].name_index = in.readChar();
            local_variable_table[i].descriptor_index = in.readChar();
            local_variable_table[i].index = in.readChar();
        }
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2 + (local_variable_table_length * 10);
        super.serialize(out);
        out.writeChar(local_variable_table_length);
        for (int i=0; i<local_variable_table_length; i++){
            out.writeChar(local_variable_table[i].start_pc);
            out.writeChar(local_variable_table[i].length);
            out.writeChar(local_variable_table[i].name_index);
            out.writeChar(local_variable_table[i].descriptor_index);
            out.writeChar(local_variable_table[i].index);
        }
    }

}
```

A34. **method_info.java**

Convience class for representing method_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** This follows the method_info in the JVM specification.
 */
public final class method_info {

    public int access_flags;
    public int name_index;
    public int descriptor_index;
    public int attributes_count;
    public attribute_info[] attributes;

    /** Constructor. Creates a method_info, initializes it with
```

```
     *  the flags set, and the name and descriptor indexes given.
     *  A new uninitialized code attribute is also created, and stored
     *  in the <i>code</i> variable.*/
    public method_info(ClassFile cf, int flags, int ni, int di,
                       int ac, attribute_info[] a) {
        access_flags = flags;
        name_index = ni;
        descriptor_index = di;
        attributes_count = ac;
        attributes = a;
    }

    /** This method creates a method_info from the current pointer in the
     *  data stream. Only called by during the serialization of a complete
     *  ClassFile from a bytestream, not normally invoked directly.
     */
    method_info(ClassFile cf, DataInputStream in)
        throws IOException{
        access_flags = in.readChar();
        name_index = in.readChar();
        descriptor_index = in.readChar();
        attributes_count = in.readChar();
        attributes = new attribute_info[attributes_count];
        for (int i=0; i<attributes_count; i++){
            in.mark(2);
            String s = cf.constant_pool[in.readChar()].toString();
            in.reset();
            if (s.equals("Code"))
                attributes[i] = new Code_attribute(cf, in);
            else if (s.equals("Exceptions"))
                attributes[i] = new Exceptions_attribute(cf, in);
            else if (s.equals("Synthetic"))
                attributes[i] = new Synthetic_attribute(cf, in);
            else if (s.equals("Deprecated"))
                attributes[i] = new Deprecated_attribute(cf, in);
            else
                attributes[i] = new attribute_info.Unknown(cf, in);
        }
    }

    /** Output serialization of the method_info to a byte array.
     *  Not normally invoked directly.
    */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeChar(access_flags);
        out.writeChar(name_index);
        out.writeChar(descriptor_index);
        out.writeChar(attributes_count);
        for (int i=0; i<attributes_count; i++)
            attributes[i].serialize(out);
    }

}
```

A35. **SourceFile_attribute.java**

Convience class for representing SourceFile_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** A SourceFile attribute is an optional fixed_length attribute in
 *  the attributes table. Only located in the ClassFile struct only
```

```
     *  once.
     */
    public final class SourceFile_attribute extends attribute_info{

        public int sourcefile_index;

        public SourceFile_attribute(ClassFile cf, int ani, int al, int sfi){
            super(cf);
            attribute_name_index = ani;
            attribute_length = al;
            sourcefile_index = sfi;
        }

        /** Used during input serialization by ClassFile only. */
        SourceFile_attribute(ClassFile cf, DataInputStream in)
            throws IOException{
            super(cf, in);
            sourcefile_index = in.readChar();
        }

        /** Used during output serialization by ClassFile only. */
        void serialize(DataOutputStream out)
            throws IOException{
            attribute_length = 2;
            super.serialize(out);
            out.writeChar(sourcefile_index);
        }

    }
```

A36. **Synthetic_attribute.java**

Convience class for representing Synthetic_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** A synthetic attribute indicates that this class does not have
 *  a generated code source. It is likely to imply that the code
 *  is generated by machine means rather than coded directly. This
 *  attribute can appear in the classfile, method_info or field_info.
 *  It is fixed length.
 */
public final class Synthetic_attribute extends attribute_info{

    public Synthetic_attribute(ClassFile cf, int ani, int al){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
    }

    /** Used during output serialization by ClassFile only. */
    Synthetic_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
    }

}
```

ANNEXURE B

B1

Method <clinit>

0 new #2 <Class test>

```
 3 dup
 4 invokespecial #3 <Method test()>
 7 putstatic #4 <Field test thisTest>
10 return
B2
Method <clinit>
 0 invokestatic #3 <Method boolean isAlreadyLoaded()>
 3 ifeq 7
 6 return
 7 new #5 <Class test>
10 dup
11 invokespecial #6 <Method test()>
14 putstatic #7 <Field test thisTest>
17 return
B3
Method <init>
 0 aload_0
 1 invokespecial #1 <Method java.lang.Object()>
 4 aload_0
 5 invokestatic #2 <Method long currentTimeMillis()>
 8 putfield #3 <Field long timestamp>
11 return
B4
Method <init>
 0 aload_0
 1 invokespecial #1 <Method java.lang.Object()>
 4 invokestatic #2 <Method boolean isAlreadyLoaded()>
 7 ifeq 11
10 return
11 aload_0
12 invokestatic #4 <Method long currentTimeMillis()>
15 putfield #5 <Field long timestamp>
18 return
B5
Method <clinit>
 0 ldc #2 <String "test">
 2 invokestatic #3 <Method boolean isAlreadyLoaded(java.lang.String)>
 5 ifeq 9
 8 return
 9 new #5 <Class test>
12 dup
13 invokespecial #6 <Method test()>
16 putstatic #7 <Field test thisTest>
19 return
B6
Method <init>
 0 aload_0
 1 invokespecial #1 <Method java.lang.Object()>
```

```
4 aload_0
5 invokestatic #2 <Method boolean isAlreadyLoaded(java.lang.Object)>
8 ifeq 12
11 return
12 aload_0
13 invokestatic #4 <Method long currentTimeMillis()>
16 putfield #5 <Field long timestamp>
19 return
```

ANNEXURE B7

This excerpt is the source-code of InitClient, which queries an "initialisation server" for the initialisation status of the relevant class or object.

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class InitClient{

   /** Protocol specific values. */
   public final static int CLOSE = -1;
   public final static int NACK = 0;
   public final static int ACK = 1;
   public final static int INITIALIZE_CLASS = 10;
   public final static int INITIALIZE_OBJECT = 20;

   /** InitServer network values. */
   public final static String serverAddress =
      System.getProperty("InitServer_network_address");
   public final static int serverPort =
      Integer.parseInt(System.getProperty("InitServer_network_port"));

   /** Table of global ID's for local objects. (hashcode-to-globalID
       mappings) */
   public final static Hashtable hashCodeToGlobalID = new Hashtable();

   /** Called when a object is being initialized. */
   public static boolean isAlreadyLoaded(Object o){

      // First of all, we need to resolve the globalID
      // for object 'o'.  To do this we use the hashCodeToGlobalID
      // table.
      int globalID = ((Integer) hashCodeToGlobalID.get(o)).intValue();

      try{

         // Next, we want to connect to the InitServer, which will inform us
         // of the initialization status of this object.
         Socket socket = new Socket(serverAddress, serverPort);
         DataOutputStream out =
            new DataOutputStream(socket.getOutputStream());
         DataInputStream in =
            new DataInputStream(socket.getInputStream());

         // Ok, now send the serialized request to the InitServer.
         out.writeInt(INITIALIZE_OBJECT);
         out.writeInt(globalID);
         out.flush();

         // Now wait for the reply.
         int status = in.readInt();      // This is a blocking call.  So we
                                         // will wait until the remote side
```

```
                                        // sends something.

            if (status == NACK){
               throw new AssertionError(
                  "Negative acknowledgement. Request failed.");
            }else if (status != ACK){
               throw new AssertionError("Unknown acknowledgement: "
                  + status + ". Request failed.");
            }

            // Next, read in a 32bit argument which is the count of previous
            // initializations.
            int count = in.readInt();

            // If the count is equal to 0, then this is the first
            // initialization, and hence isAlreadyLoaded should be false.
            // If however, the count is greater than 0, then this is already
            // initialized, and thus isAlreadyLoaded should be true.
            boolean isAlreadyLoaded = (count == 0 ? false : true);

            // Close down the connection.
            out.writeInt(CLOSE);
            out.flush();
            out.close();
            in.close();

            socket.close();          // Make sure to close the socket.

            // Return the value of the isAlreadyLoaded variable.
            return isAlreadyLoaded;

         }catch (IOException e){
            throw new AssertionError("Exception: " + e.toString());
         }
      }

      /** Called when a class is being initialized. */
      public static boolean isAlreadyLoaded(String name){

         try{

            // First of all, we want to connect to the InitServer, which will
            // inform us of the initialization status of this class.
            Socket socket = new Socket(serverAddress, serverPort);
            DataOutputStream out =
               new DataOutputStream(socket.getOutputStream());
            DataInputStream in =
               new DataInputStream(socket.getInputStream());

            // Ok, now send the serialized request to the InitServer.
            out.writeInt(INITIALIZE_CLASS);
            out.writeInt(name.length());          // A 32bit length argument of
                                                  // the String name.
            out.write(name.getBytes(), 0, name.length());      // The byte-
                                                               // encoded
                                                               // String name.
            out.flush();

            // Now wait for the reply.
            int status = in.readInt();      // This is a blocking call. So we
                                            // will wait until the remote side
                                            // sends something.

            if (status == NACK){
               throw new AssertionError(
```

```
                "Negative acknowledgement. Request failed.");
            }else if (status != ACK){
               throw new AssertionError("Unknown acknowledgement: "
                  + status + ". Request failed.");
            }

            // Next, read in a 32bit argument which is the count of the
            // previous intializations.
            int count = in.readInt();

            // If the count is equal to 0, then this is the first
            // initialization, and hence isAlreadyLoaded should be false.
            // If however, the count is greater than 0, then this is already
            // loaded, and thus isAlreadyLoaded should be true.
            boolean isAlreadyLoaded = (count == 0 ? false : true);

            // Close down the connection.
            out.writeInt(CLOSE);
            out.flush();
            out.close();
            in.close();

            socket.close();          // Make sure to close the socket.

            // Return the value of the isAlreadyLoaded variable.
            return isAlreadyLoaded;

         }catch (IOException e){
            throw new AssertionError("Exception: " + e.toString());
         }
      }

}
```

ANNEXURE B8

This excerpt is the source-code of InitServer, which receives an initialisation status query by InitClient and in response returns the corresponding status.

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class InitServer implements Runnable{

   /** Protocol specific values */
   public final static int CLOSE = -1;
   public final static int NACK = 0;
   public final static int ACK = 1;
   public final static int INITIALIZE_CLASS = 10;
   public final static int INITIALIZE_OBJECT= 20;
   /** InitServer network values. */
   public final static int serverPort = 20001;
   /** Table of initialization records. */
   public final static Hashtable initializations = new Hashtable();

   /** Private input/output objects. */
   private Socket socket = null;
   private DataOutputStream outputStream;
   private DataInputStream inputStream;
   private String address;

   public static void main(String[] s)
   throws Exception{
```

```
        System.out.println("InitServer_network_address="
           + InetAddress.getLocalHost().getHostAddress());
        System.out.println("InitServer_network_port=" + serverPort);

        // Create a serversocket to accept incoming initialization operation
        // connections.
        ServerSocket serverSocket = new ServerSocket(serverPort);

        while (!Thread.interrupted()){

           // Block until an incoming initialization operation connection.
           Socket socket = serverSocket.accept();

           // Create a new instance of InitServer to manage this
           // initialization operation connection.
           new Thread(new InitServer(socket)).start();

        }

     }

     /** Constructor. Initialize this new InitServer instance with necessary
         resources for operation. */
     public InitServer(Socket s){
        socket = s;
        try{
           outputStream = new DataOutputStream(s.getOutputStream());
           inputStream = new DataInputStream(s.getInputStream());
           address = s.getInetAddress().getHostAddress();
        }catch (IOException e){
           throw new AssertionError("Exception: " + e.toString());
        }
     }
     /** Main code body. Decode incoming initialization operation requests and
         execute accordingly. */
     public void run(){

        try{

           // All commands are implemented as 32bit integers.
           // Legal commands are listed in the "protocol specific values"
           // fields above.
           int command = inputStream.readInt();

           // Continue processing commands until a CLOSE operation.
           while (command != CLOSE){

              if (command == INITIALIZE_CLASS){          // This is an
                                                         // INITIALIZE_CLASS
                                                         // operation.

                 // Read in a 32bit length field 'l', and a String name for
                 // this class of length 'l'.
                 int length = inputStream.readInt();
                 byte[] b = new byte[length];
                 inputStream.read(b, 0, b.length);
                 String className = new String(b, 0, length);

                 // Synchronize on the initializations table in order to
                 // ensure thread-safety.
                 synchronized (initializations){

                    // Locate the previous initializations entry for this
                    // class, if any.
                    Integer entry = (Integer) initializations.get(className);
```

```
            if (entry == null){    // This is an unknown class so
                                   // update the table with a
                                   // corresponding entry.

               initializations.put(className, new Integer(1));

               // Send a positive acknowledgement to InitClient,
               // together with the count of previous initializations
               // of this class - which in this case of an unknown
               // class must be 0.
               outputStream.writeInt(ACK);
               outputStream.writeInt(0);
               outputStream.flush();

            }else{                 // This is a known class, so update
                                   // the count of initializations.

               initializations.put(className,
                  new Integer(entry.intValue() + 1));

               // Send a positive acknowledgement to InitClient,
               // together with the count of previous initializtions
               // of this class - which in this case of a known class
               // must be the value of "entry.intValue()".
               outputStream.writeInt(ACK);
               outputStream.writeInt(entry.intValue());
               outputStream.flush();

            }

         }

      }else if (command == INITIALIZE_OBJECT){ // This is an
                                               // INITIALIZE_OBJECT
                                               // operation.

         // Read in the globalID of the object to be initialized.
         int globalID = inputStream.readInt();

         // Synchronize on the initializations table in order to
         // ensure thread-safety.
         synchronized (initializations){

            // Locate the previous initializations entry for this
            // object, if any.
            Integer entry = (Integer) initializations.get(
               new Integer(globalID));

            if (entry == null){    // This is an unknown object so
                                   // update the table with a
                                   // corresponding entry.

               initializations.put(new Integer(globalID),
                  new Integer(1));

               // Send a positive acknowledgement to InitClient,
               // together with the count of previous initializations
               // of this object - which in this case of an unknown
               // object must be 0.
               outputStream.writeInt(ACK);
               outputStream.writeInt(0);
               outputStream.flush();

            }else{          // This is a known object so update the
```

```
                                    // count of initializations.

                        initializations.put(new Integer(globalID),
                           new Integer(entry.intValue() + 1));

                        // Send a positive acknowledgement to InitClient,
                        // together with the count of previous initializations
                        // of this object - which in this case of a known
                        // object must be value "entry.intValue()".
                        outputStream.writeInt(ACK);
                        outputStream.writeInt(entry.intValue());
                        outputStream.flush();

                     }

                  }

               }else{         // Unknown command.
                  throw new AssertionError(
                     "Unknown command. Operation failed.");
               }

               // Read in the next command.
               command = inputStream.readInt();

            }

         }catch (Exception e){
            throw new AssertionError("Exception: " + e.toString());
         }finally{
            try{
               // Closing down. Cleanup this connection.
               outputStream.flush();
               outputStream.close();
               inputStream.close();
               socket.close();
            }catch (Throwable t){
               t.printStackTrace();
            }
            // Garbage these references.
            outputStream = null;
            inputStream = null;
            socket = null;
         }
      }

   }
```

ANNEXURE B9

This excerpt is the source-code of the example application used in the before/after examples of Annexure B.

```
import java.lang.*;

public class example{

   /** Shared static field. */
   public static example currentExample;

   /** Shared instance field. */
   public long timestamp;

   /** Static intializer. (clinit) */
   static{
```

```
        currentExample = new example();

    }

    /** Instance intializer (init) */
    public example(){

        timestamp = System.currentTimeMillis();

    }

}
```

ANNEXURE B10

InitLoader.java

This excerpt is the source-code of InitLoader, which modifies an application as it is being loaded.

```
import java.lang.*;
import java.io.*;
import java.net.*;

public class InitLoader extends URLClassLoader{

    public InitLoader(URL[] urls){
        super(urls);
    }

    protected Class findClass(String name)
    throws ClassNotFoundException{

        ClassFile cf = null;

        try{
            BufferedInputStream in = new
                BufferedInputStream(findResource(name.replace('.',
                '/').concat(".class")).openStream());

            cf = new ClassFile(in);

        }catch (Exception e){throw new ClassNotFoundException(e.toString());}

        for (int i=0; i<cf.methods_count; i++){

            // Find the <clinit> method_info struct.
            String methodName = cf.constant_pool[
                cf.methods[i].name_index].toString();
            if (!methodName.equals("<clinit>")){
                continue;
            }

            // Now find the Code_attribute for the <clinit> method.
            for (int j=0; j<cf.methods[i].attributes_count; j++){
                if (!(cf.methods[i].attributes[j] instanceof Code_attribute))
                    continue;

                Code_attribute ca = (Code_attribute) cf.methods[i].attributes[j];

                // First, shift the code[] down by 4 instructions.
                byte[][] code2 = new byte[ca.code.length+4][];
                System.arraycopy(ca.code, 0, code2, 4, ca.code.length);
                ca.code = code2;
```

```
            // Then enlarge the constant_pool by 7 items.
            cp_info[] cpi = new cp_info[cf.constant_pool.length+7];
            System.arraycopy(cf.constant_pool, 0, cpi, 0,
               cf.constant_pool.length);
            cf.constant_pool = cpi;
            cf.constant_pool_count += 7;

            // Now add the constant pool items for these instructions, starting
            // with String.
            CONSTANT_String_info csi = new CONSTANT_String_info(
        ((CONSTANT_Class_info)cf.constant_pool[cf.this_class]).name_index);
            cf.constant_pool[cf.constant_pool.length-7] = csi;

            // Now add the UTF for class.
            CONSTANT_Utf8_info u1 = new CONSTANT_Utf8_info("InitClient");
            cf.constant_pool[cf.constant_pool.length-6] = u1;

            // Now add the CLASS for the previous UTF.
            CONSTANT_Class_info c1 =
               new CONSTANT_Class_info(cf.constant_pool.length-6);
            cf.constant_pool[cf.constant_pool.length-5] = c1;

            // Next add the first UTF for NameAndType.
            u1 = new CONSTANT_Utf8_info("isAlreadyLoaded");
            cf.constant_pool[cf.constant_pool.length-4] = u1;

            // Next add the second UTF for NameAndType.
            u1 = new CONSTANT_Utf8_info("(Ljava/lang/String;)Z");
            cf.constant_pool[cf.constant_pool.length-3] = u1;

            // Next add the NameAndType for the previous two UTFs.
            CONSTANT_NameAndType_info n1 = new CONSTANT_NameAndType_info(
               cf.constant_pool.length-4, cf.constant_pool.length-3);
            cf.constant_pool[cf.constant_pool.length-2] = n1;

            // Next add the Methodref for the previous CLASS and NameAndType.
            CONSTANT_Methodref_info m1 = new CONSTANT_Methodref_info(
               cf.constant_pool.length-5, cf.constant_pool.length-2);
            cf.constant_pool[cf.constant_pool.length-1] = m1;

            // Now with that done, add the instructions into the code, starting
            // with LDC.
            ca.code[0] = new byte[3];
            ca.code[0][0] = (byte) 19;
            ca.code[0][1] = (byte) (((cf.constant_pool.length-7) >> 8) & 0xff);
            ca.code[0][2] = (byte) ((cf.constant_pool.length-7) & 0xff);

            // Now Add the INVOKESTATIC instruction.
            ca.code[1] = new byte[3];
            ca.code[1][0] = (byte) 184;
            ca.code[1][1] = (byte) (((cf.constant_pool.length-1) >> 8) & 0xff);
            ca.code[1][2] = (byte) ((cf.constant_pool.length-1) & 0xff);

            // Next add the IFEQ instruction.
            ca.code[2] = new byte[3];
            ca.code[2][0] = (byte) 153;
            ca.code[2][1] = (byte) ((4 >> 8) & 0xff);
            ca.code[2][2] = (byte) (4 & 0xff);

            // Finally, add the RETURN instruction.
            ca.code[3] = new byte[1];
            ca.code[3][0] = (byte) 177;

            // Lastly, increment the CODE_LENGTH and ATTRIBUTE_LENGTH values.
```

```
            ca.code_length += 10;
            ca.attribute_length += 10;

        }
    }

    try{
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        cf.serialize(out);

        byte[] b = out.toByteArray();

        return defineClass(name, b, 0, b.length);

    }catch (Exception e){
     e.printStackTrace();
        throw new ClassNotFoundException(name);
    }

  }
}
```

Annexure C

C1. TYPICAL PRIOR ART FINALIZATION FOR A SINGLE MACHINE:

Method finalize()
0 getstatic #9 <Field java.io.PrintStream out>
3 ldc #24 <String "Deleted...">
5 invokevirtual #16 <Method void println(java.lang.String)>
8 return

C2. PREFERRED FINALIZATION FOR MULTIPLE MACHINES

Method finalize()
0 invokestatic #3 <Method boolean isLastReference()>
3 ifne 7
6 return
7 getstatic #9 <Field java.io.PrintStream out>
10 ldc #24 <String "Deleted...">
12 invokevirtual #16 <Method void println(java.lang.String)>
15 return

C3. PREFERRED FINALIZATION FOR MULTIPLE MACHINES (Alternative)

Method finalize()
0 aload_0
1 invokestatic #3 <Method boolean isLastReference(java.lang.Object)>
4 ifne 8
7 return
8 getstatic #9 <Field java.io.PrintStream out>
11 ldc #24 <String "Deleted...">
13 invokevirtual #16 <Method void println(java.lang.String)>
16 return

Annexure C4

```
import java.lang.*;

public class example{
```

```
   /** Finalize method. */
   protected void finalize() throws Throwable{

      // "Deleted..." is printed out when this object is garbaged.
      System.out.println("Deleted...");

   }
}
```

Annexure C5

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class FinalClient{

   /** Protocol specific values. */
   public final static int CLOSE = -1;
   public final static int NACK = 0;
   public final static int ACK = 1;
   public final static int FINALIZE_OBJECT = 10;

   /** FinalServer network values. */
   public final static String serverAddress =
      System.getProperty("FinalServer_network_address");
   public final static int serverPort =
      Integer.parseInt(System.getProperty("FinalServer_network_port"));

   /** Table of global ID's for local objects. (hashcode-to-globalID
       mappings) */
   public final static Hashtable hashCodeToGlobalID = new Hashtable();

   /** Called when a object is being finalized. */
   public static boolean isLastReference(Object o){

      // First of all, we need to resolve the globalID for object 'o'.
      // To do this we use the hashCodeToGlobalID table.
      int globalID = ((Integer) hashCodeToGlobalID.get(o)).intValue();

      try{

         // Next, we want to connect to the FinalServer, which will inform
         // us of the finalization status of this object.
         Socket socket = new Socket(serverAddress, serverPort);
         DataOutputStream out =
            new DataOutputStream(socket.getOutputStream());
         DataInputStream in = new DataInputStream(socket.getInputStream());

         // Ok, now send the serialized request to the FinalServer.
         out.writeInt(FINALIZE_OBJECT);
         out.writeInt(globalID);
         out.flush();

         // Now wait for the reply.
         int status = in.readInt();     // This is a blocking call.  So we
                                        // will wait until the remote side
                                        // sends something.

         if (status == NACK){
            throw new AssertionError(
               "Negative acknowledgement. Request failed.");
         }else if (status != ACK){
            throw new AssertionError("Unknown acknowledgement: "
```

```
                + status + ". Request failed.");
            }

            // Next, read in a 32bit argument which is the count of the
            // remaining finalizations
            int count = in.readInt();

            // If the count is equal to 1, then this is the last finalization,
            // and hence isLastReference should be true.
            // If however, the count is greater than 1, then this is not the
            // last finalization, and thus isLastReference should be false.
            boolean isLastReference = (count == 1 ? true : false);

            // Close down the connection.
            out.writeInt(CLOSE);
            out.flush();
            out.close();
            in.close();

            socket.close();          // Make sure to close the socket.

            // Return the value of the isLastReference variable.
            return isLastReference;

        }catch (IOException e){
            throw new AssertionError("Exception: " + e.toString());
        }
    }
}
```

Annexure C6

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class FinalServer implements Runnable{

    /** Protocol specific values */
    public final static int CLOSE = -1;
    public final static int NACK = 0;
    public final static int ACK = 1;
    public final static int FINALIZE_OBJECT = 10;

    /** FinalServer network values. */
    public final static int serverPort = 20001;

    /** Table of finalization records. */
    public final static Hashtable finalizations = new Hashtable();

    /** Private input/output objects. */
    private Socket socket = null;
    private DataOutputStream outputStream;
    private DataInputStream inputStream;
    private String address;

    public static void main(String[] s)
    throws Exception{

        System.out.println("FinalServer network address="
            + InetAddress.getLocalHost().getHostAddress());
        System.out.println("FinalServer network port=" + serverPort);

        // Create a serversocket to accept incoming initialization operation
        // connections.
```

```
      ServerSocket serverSocket = new ServerSocket(serverPort);

      while (!Thread.interrupted()){

         // Block until an incoming initialization operation connection.
         Socket socket = serverSocket.accept();

         // Create a new instance of InitServer to manage this
         // initialization operation connection.
         new Thread(new FinalServer(socket)).start();

      }

   }

   /** Constructor. Initialize this new FinalServer instance with necessary
       resources for operation. */
   public FinalServer(Socket s){
      socket = s;
      try{
         outputStream = new DataOutputStream(s.getOutputStream());
         inputStream = new DataInputStream(s.getInputStream());
         address = s.getInetAddress().getHostAddress();
      }catch (IOException e){
         throw new AssertionError("Exception: " + e.toString());
      }
   }

   /** Main code body. Decode incoming finalization operation requests and
       execute accordingly. */
   public void run(){

      try{

         // All commands are implemented as 32bit integers.
         // Legal commands are listed in the "protocol specific values"
         // fields above.
         int command = inputStream.readInt();

         // Continue processing commands until a CLOSE operation.
         while (command != CLOSE){

            if (command == FINALIZE_OBJECT){          // This is a
                                                      // FINALIZE_OBJECT
                                                      // operation.

               // Read in the globalID of the object to be finalized.
               int globalID = inputStream.readInt();

               // Synchronize on the finalizations table in order to ensure
               // thread-safety.
               synchronized (finalizations){

                  // Locate the previous finalizations entry for this
                  // object, if any.
                  Integer entry = (Integer) finalizations.get(
                     new Integer(globalID));

                  if (entry == null){
                     throw new AssertionError("Unknown object.");
                  }else if (entry.intValue() < 1){
                     throw new AssertionError("Invalid count.");
                  }else if (entry.intValue() == 1){   // Count of 1 means
                                                      // this is the last
                                                      // reference, hence
```

```
                                                    // remove from table.
                        finalizations.remove(new Integer(globalID));

                        // Send a positive acknowledgement to FinalClient,
                        // together with the count of remaining references -
                        // which in this case is 1.
                        outputStream.writeInt(ACK);
                        outputStream.writeInt(1);
                        outputStream.flush();

                     }else{          // This is not the last remaining
                                     // reference, as count is greater than 1.
                                     // Decrement count by 1.

                        finalizations.put(new Integer(globalID),
                           new Integer(entry.intValue() - 1));

                        // Send a positive acknowledgement to FinalClient,
                        // together with the count of remaining references to
                        // this object - which in this case of must be value
                        // "entry.intValue()".
                        outputStream.writeInt(ACK);
                        outputStream.writeInt(entry.intValue());
                        outputStream.flush();

                     }

                  }

               }else{          // Unknown command.
                  throw new AssertionError(
                     "Unknown command. Operation failed.");
               }

               // Read in the next command.
               command = inputStream.readInt();

            }

         }catch (Exception e){
            throw new AssertionError("Exception: " + e.toString());
         }finally{
            try{
               // Closing down. Cleanup this connection.
               outputStream.flush();
               outputStream.close();
               inputStream.close();
               socket.close();
            }catch (Throwable t){
               t.printStackTrace();
            }
            // Garbage these references.
            outputStream = null;
            inputStream = null;
            socket = null;
         }
      }

   }
```

ANNEXURE C7

FinalLoader.java

This excerpt is the source-code of FinalLoader, which modifies an application as it is being loaded.

```
import java.lang.*;
import java.io.*;
import java.net.*;

public class FinalLoader extends URLClassLoader{

   public FinalLoader(URL[] urls){
      super(urls);
   }

   protected Class findClass(String name)
   throws ClassNotFoundException{

      ClassFile cf = null;

      try{
         BufferedInputStream in =
            new BufferedInputStream(findResource(name.replace('.',
            '/').concat(".class")).openStream());

         cf = new ClassFile(in);

      }catch (Exception e){throw new ClassNotFoundException(e.toString());}

      for (int i=0; i<cf.methods_count; i++){

         // Find the finalize method_info struct.
         String methodName = cf.constant_pool[
            cf.methods[i].name_index].toString();
         if (!methodName.equals("finalize")){
            continue;
         }

         // Now find the Code_attribute for the finalize method.
         for (int j=0; j<cf.methods[i].attributes_count; j++){
            if (!(cf.methods[i].attributes[j] instanceof Code_attribute))
               continue;

            Code_attribute ca = (Code_attribute) cf.methods[i].attributes[j];

            // First, shift the code[] down by 4 instructions.
            byte[][] code2 = new byte[ca.code.length+4][];
            System.arraycopy(ca.code, 0, code2, 4, ca.code.length);
            ca.code = code2;

            // Then enlarge the constant_pool by 6 items.
            cp_info[] cpi = new cp_info[cf.constant_pool.length+6];
            System.arraycopy(cf.constant_pool, 0, cpi, 0,
               cf.constant_pool.length);
            cf.constant_pool = cpi;
            cf.constant_pool_count += 6;

            // Now add the UTF for class.
            CONSTANT_Utf8_info u1 = new CONSTANT_Utf8_info("FinalClient");
            cf.constant_pool[cf.constant_pool.length-6] = u1;

            // Now add the CLASS for the previous UTF.
            CONSTANT_Class_info c1 =
               new CONSTANT_Class_info(cf.constant_pool.length-6);
            cf.constant_pool[cf.constant_pool.length-5] = c1;

            // Next add the first UTF for NameAndType.
```

```
            u1 = new CONSTANT_Utf8_info("isLastReference");
            cf.constant_pool[cf.constant_pool.length-4] = u1;

            // Next add the second UTF for NameAndType.
            u1 = new CONSTANT_Utf8_info("(Ljava/lang/Object;)Z");
            cf.constant_pool[cf.constant_pool.length-3] = u1;

            // Next add the NameAndType for the previous two UTFs.
            CONSTANT_NameAndType_info n1 = new CONSTANT_NameAndType_info(
               cf.constant_pool.length-4, cf.constant_pool.length-3);
            cf.constant_pool[cf.constant_pool.length-2] = n1;

            // Next add the Methodref for the previous CLASS and NameAndType.
            CONSTANT_Methodref_info m1 = new CONSTANT_Methodref_info(
               cf.constant_pool.length-5, cf.constant_pool.length-2);
            cf.constant_pool[cf.constant_pool.length-1] = m1;

            // Now with that done, add the instructions into the code, starting
            // with LDC.
            ca.code[0] = new byte[1];
            ca.code[0][0] = (byte) 42;

            // Now Add the INVOKESTATIC instruction.
            ca.code[1] = new byte[3];
            ca.code[1][0] = (byte) 184;
            ca.code[1][1] = (byte) (((cf.constant_pool.length-1) >> 8) & 0xff);
            ca.code[1][2] = (byte) ((cf.constant_pool.length-1) & 0xff);

            // Next add the IFNE instruction.
            ca.code[2] = new byte[3];
            ca.code[2][0] = (byte) 154;
            ca.code[2][1] = (byte) ((4 >> 8) & 0xff);
            ca.code[2][2] = (byte) (4 & 0xff);

            // Finally, add the RETURN instruction.
            ca.code[3] = new byte[1];
            ca.code[3][0] = (byte) 177;

            // Lastly, increment the CODE_LENGTH and ATTRIBUTE_LENGTH values.
            ca.code_length += 8;
            ca.attribute_length += 8;

         }
      }

      try{

         ByteArrayOutputStream out = new ByteArrayOutputStream();
         cf.serialize(out);

         byte[] b = out.toByteArray();

         return defineClass(name, b, 0, b.length);

      }catch (Exception e){
       e.printStackTrace();
         throw new ClassNotFoundException(name);
      }

   }

}
```

Annexure D1

```
Method void run()
  0 getstatic #2 <Field java.lang.Object LOCK>
  3 dup
  4 astore_1
  5 monitorenter
  6 getstatic #3 <Field int counter>
  9 iconst_1
 10 iadd
 11 putstatic #3 <Field int counter>
 14 aload_1
 15 monitorexit
 16 return
```

Annexure D2

```
Method void run()
  0 getstatic #2 <Field java.lang.Object LOCK>
  3 dup
  4 astore_1
  5 dup
  6 monitorenter
  7 invokestatic #23 <Method void acquireLock(java.lang.Object)>
 10 getstatic #3 <Field int counter>
 13 iconst_1
 14 iadd
 15 putstatic #3 <Field int counter>
 18 aload_1
 19 dup
 20 invokestatic #24 <Method void releaseLock(java.lang.Object)>
 23 monitorexit
 24 return
```

Annexure D3

```
import java.lang.*;

public class example{

   /** Shared static field. */
   public final static Object LOCK = new Object();

   /** Shared static field. */
   public static int counter = 0;

   /** Example method using synchronization. This method serves to
       illustrate the use of synchronization to implement thread-safe
       modification of a shared memory location by potentially multiple
       threads. */
   public void run(){

      // First acquire the lock, otherwise any memory writes we do will be
      // prone to race-conditions.
      synchronized (LOCK){

         // Now that we have acquired the lock, we can safely modify memory
         // in a thread-safe manner.
         counter++;
```

```
        }

    }

}
```

Annexure D4

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class LockClient{

   /** Protocol specific values. */
   public final static int CLOSE = -1;
   public final static int NACK = 0;
   public final static int ACK = 1;
   public final static int ACQUIRE_LOCK = 10;
   public final static int RELEASE_LOCK = 20;
   /** LockServer network values. */
   public final static String serverAddress =
      System.getProperty("LockServer_network_address");
   public final static int serverPort =
      Integer.parseInt(System.getProperty("LockServer_network_port"));

   /** Table of global ID's for local objects. (hashcode-to-globalID
       mappings) */
   public final static Hashtable hashCodeToGlobalID = new Hashtable();

   /** Called when an application is to acquire a lock. */
   public static void acquireLock(Object o){

      // First of all, we need to resolve the globalID for object 'o'.
      // To do this we use the hashCodeToGlobalID table.
      int globalID = ((Integer) hashCodeToGlobalID.get(o)).intValue();

      try{

         // Next, we want to connect to the LockServer, which will grant us
         // the global lock.
         Socket socket = new Socket(serverAddress, serverPort);
         DataOutputStream out =
            new DataOutputStream(socket.getOutputStream());
         DataInputStream in = new DataInputStream(socket.getInputStream());

         // Ok, now send the serialized request to the lock server.
         out.writeInt(ACQUIRE_LOCK);
         out.writeInt(globalID);
         out.flush();

         // Now wait for the reply.
         int status = in.readInt();     // This is a blocking call.  So we
                                        // will wait until the remote side
                                        // sends something.

         if (status == NACK){
            throw new AssertionError(
               "Negative acknowledgement. Request failed.");
         }else if (status != ACK){
            throw new AssertionError("Unknown acknowledgement: "
               + status + ". Request failed.");
         }
```

```
            // Close down the connection.
            out.writeInt(CLOSE);
            out.flush();
            out.close();
            in.close();

            socket.close();          // Make sure to close the socket.

            // This is a good acknowledgement, thus we can return now because
            // global lock is now acquired.
            return;

         }catch (IOException e){
            throw new AssertionError("Exception: " + e.toString());
         }

      }

      /** Called when an application is to release a lock. */
      public static void releaseLock(Object o){

         // First of all, we need to resolve the globalID for object 'o'.
         // To do this we use the hashCodeToGlobalID table.
         int globalID = ((Integer) hashCodeToGlobalID.get(o)).intValue();

         try{

            // Next, we want to connect to the LockServer, which records us as
            // the owner of the global lock for object 'o'.
            Socket socket = new Socket(serverAddress, serverPort);
            DataOutputStream out =
               new DataOutputStream(socket.getOutputStream());
            DataInputStream in = new DataInputStream(socket.getInputStream());

            // Ok, now send the serialized request to the lock server.
            out.writeInt(RELEASE_LOCK);
            out.writeInt(globalID);
            out.flush();

            // Now wait for the reply.
            int status = in.readInt();      // This is a blocking call.  So we
                                            // will wait until the remote side
                                            // sends something.

            if (status == NACK){
               throw new AssertionError(
                  "Negative acknowledgement. Request failed.");
            }else if (status != ACK){
               throw new AssertionError("Unknown acknowledgement: "
                  + status + ". Request failed.");
            }

            // Close down the connection.
            out.writeInt(CLOSE);
            out.flush();
            out.close();
            in.close();

            socket.close();          // Make sure to close the socket.

            // This is a good acknowledgement, return because global lock is
            // now released.
            return;
```

```
        }catch (IOException e){
           throw new AssertionError("Exception: " + e.toString());
        }

    }

}
```

Annexure D5

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class LockServer implements Runnable{

   /** Protocol specific values */
   public final static int CLOSE = -1;
   public final static int NACK = 0;
   public final static int ACK = 1;
   public final static int ACQUIRE_LOCK = 10;
   public final static int RELEASE_LOCK = 20;

   /** LockServer network values. */
   public final static int serverPort = 20001;

   /** Table of lock records. */
   public final static Hashtable locks = new Hashtable();

   /** Linked list of waiting LockManager objects. */
   public LockServer next = null;

   /** Address of remote LockClient. */
   public final String address;

   /** Private input/output objects. */
   private Socket socket = null;
   private DataOutputStream outputStream;
   private DataInputStream inputStream;

   public static void main(String[] s)
   throws Exception{

      System.out.println("LockServer network address="
         + InetAddress.getLocalHost().getHostAddress());
      System.out.println("LockServer network port=" + serverPort);

      // Create a serversocket to accept incoming lock operation
      // connections.
      ServerSocket serverSocket = new ServerSocket(serverPort);

      while (!Thread.interrupted()){

         // Block until an incoming lock operation connection.
         Socket socket = serverSocket.accept();

         // Create a new instance of LockServer to manage this lock
         // operation connection.
         new Thread(new LockServer(socket)).start();

      }

   }
```

```
/** Constructor. Initialise this new LockServer instance with necessary
    resources for operation. */
public LockServer(Socket s){
   socket = s;
   try{
      outputStream = new DataOutputStream(s.getOutputStream());
      inputStream = new DataInputStream(s.getInputStream());
      address = s.getInetAddress().getHostAddress();
   }catch (IOException e){
      throw new AssertionError("Exception: " + e.toString());
   }
}

/** Main code body. Decode incoming lock operation requests and execute
    accordingly. */
public void run(){

   try{

      // All commands are implemented as 32bit integers.
      // Legal commands are listed in the "protocol specific values"
      // fields above.
      int command = inputStream.readInt();

      // Continue processing commands until a CLOSE operation.
      while (command != CLOSE){

         if (command == ACQUIRE_LOCK){            // This is an
                                                  // ACQUIRE_LOCK
                                                  // operation.

            // Read in the globalID of the object to be locked.
            int globalID = inputStream.readInt();

            // Synchronize on the locks table in order to ensure thread-
            // safety.
            synchronized (locks){

               // Check for an existing owner of this lock.
               LockServer lock = (LockServer) locks.get(
                  new Integer(globalID));

               if (lock == null){    // No-one presently owns this lock,
                                     // so acquire it.

                  locks.put(new Integer(globalID), this);

                  acquireLock();     // Signal to the client the
                                     // successful acquisition of this
                                     // lock.

               }else{                // Already owned. Append ourselves
                                     // to end of queue.

                  // Search for the end of the queue. (Implemented as
                  // linked-list)
                  while (lock.next != null){
                     lock = lock.next;
                  }

                  lock.next = this;  // Append this lock request at end.
               }

            }
```

```
            }else if (command == RELEASE_LOCK){      // This is a
                                                     // RELEASE_LOCK
                                                     // operation.

               // Read in the globalID of the object to be locked.
               int globalID = inputStream.readInt();

               // Synchronize on the locks table in order to ensure thread-
               // safety.
               synchronized (locks){

                  // Check to make sure we are the owner of this lock.
                  LockServer lock = (LockServer) locks.get(
                     new Integer(globalID));

                  if (lock == null){
                     throw new AssertionError("Unlocked. Release failed.");
                  }else if (lock.address != this.address){
                     throw new AssertionError("Trying to release a lock "
                        + "which this client doesn't own. Release "
                        + "failed.");
                  }

                  lock = lock.next;
                  lock.acquireLock();   // Signal to the client the
                                        // successful acquisition of this
                                        // lock.

                  // Shift the linked list of pending acquisitions forward
                  // by one.
                  locks.put(new Integer(globalID), lock);

                  // Clear stale reference.
                  next = null;
               }

               releaseLock();     // Signal to the client the successful
                                  // release of this lock.

            }else{         // Unknown command.
               throw new AssertionError(
                  "Unknown command. Operation failed.");
            }

            // Read in the next command.
            command = inputStream.readInt();

         }

      }catch (Exception e){
         throw new AssertionError("Exception: " + e.toString());
      }finally{
         try{
            // Closing down. Cleanup this connection.
            outputStream.flush();
            outputStream.close();
            inputStream.close();
            socket.close();
         }catch (Throwable t){
            t.printStackTrace();
         }
         // Garbage these references.
         outputStream = null;
         inputStream = null;
         socket = null;
```

```
      }

   }

   /** Send a positive acknowledgement of an ACQUIRE_LOCK operation. */
   public void acquireLock() throws IOException{
      outputStream.writeInt(ACK);
      outputStream.flush();
   }

   /** Send a positive acknowledgement of a RELEASE_LOCK operation. */
   public void releaseLock() throws IOException{
      outputStream.writeInt(ACK);
      outputStream.flush();
   }

}
```

ANNEXURE D6

LockLoader.java

This excerpt is the source-code of LockLoader, which modifies an application as it is being loaded.

```
import java.lang.*;
import java.io.*;
import java.net.*;

public class LockLoader extends URLClassLoader{

   public LockLoader(URL[] urls){
      super(urls);
   }

   protected Class findClass(String name)
   throws ClassNotFoundException{

      ClassFile cf = null;

      try{
         BufferedInputStream in =
            new BufferedInputStream(findResource(name.replace('.',
            '/').concat(".class")).openStream());

         cf = new ClassFile(in);

      }catch (Exception e){throw new ClassNotFoundException(e.toString());}

      // Class-wide pointers to the enterindex and exitindex.
      int enterindex = -1;
      int exitindex = -1;

      for (int i=0; i<cf.methods_count; i++){
         for (int j=0; j<cf.methods[i].attributes_count; j++){
            if (!(cf.methods[i].attributes[j] instanceof Code_attribute))
               continue;

            Code_attribute ca = (Code_attribute) cf.methods[i].attributes[j];

            boolean changed = false;

            for (int z=0; z<ca.code.length; z++){
               if ((ca.code[z][0] & 0xff) == 194){    // Opcode for a
                                                      // MONITORENTER
                                                      // instruction.
```

```
                changed = true;

                // Next, realign the code array, making room for the
                // insertions.
                byte[][] code2 = new byte[ca.code.length+2][];
                System.arraycopy(ca.code, 0, code2, 0, z);
                code2[z+1] = ca.code[z];
                System.arraycopy(ca.code, z+1, code2, z+3,
                   ca.code.length-(z+1));
                ca.code = code2;


                // Next, insert the DUP instruction.
                ca.code[z] = new byte[1];
                ca.code[z][0] = (byte) 89;


                // Finally, insert the INVOKESTATIC instruction.
                if (enterindex == -1){
                   // This is the first time this class is encountering the
                   // acquirelock instruction, so have to add it to the
                   // constant pool.
                   cp_info[] cpi = new cp_info[cf.constant_pool.length+6];
                   System.arraycopy(cf.constant_pool, 0, cpi, 0,
                      cf.constant_pool.length);
                   cf.constant_pool = cpi;
                   cf.constant_pool_count += 6;

                   CONSTANT_Utf8_info u1 =
                      new CONSTANT_Utf8_info("LockClient");
                   cf.constant_pool[cf.constant_pool.length-6] = u1;

                   CONSTANT_Class_info c1 = new CONSTANT_Class_info(
                      cf.constant_pool_count-6);
                   cf.constant_pool[cf.constant_pool.length-5] = c1;

                   u1 = new CONSTANT_Utf8_info("acquireLock");
                   cf.constant_pool[cf.constant_pool.length-4] = u1;

                   u1 = new CONSTANT_Utf8_info("(Ljava/lang/Object;)V");
                   cf.constant_pool[cf.constant_pool.length-3] = u1;

                   CONSTANT_NameAndType_info n1 =
                      new CONSTANT_NameAndType_info(
                      cf.constant_pool.length-4, cf.constant_pool.length-3);
                   cf.constant_pool[cf.constant_pool.length-2] = n1;

                   CONSTANT_Methodref_info m1 = new CONSTANT_Methodref_info(
                      cf.constant_pool.length-5, cf.constant_pool.length-2);
                   cf.constant_pool[cf.constant_pool.length-1] = m1;
                   enterindex = cf.constant_pool.length-1;
                }
                ca.code[z+2] = new byte[3];
                ca.code[z+2][0] = (byte) 184;
                ca.code[z+2][1] = (byte) ((enterindex >> 8) & 0xff);
                ca.code[z+2][2] = (byte) (enterindex & 0xff);

                // And lastly, increase the CODE_LENGTH and ATTRIBUTE_LENGTH
                // values.
                ca.code_length += 4;
                ca.attribute_length += 4;

                z += 1;

             }else if ((ca.code[z][0] & 0xff) == 195){    // Opcode for a
```

```
                                                    // MONITOREXIT
                                                    // instruction.
                changed = true;

                // Next, realign the code array, making room for the
                // insertions.
                byte[][] code2 = new byte[ca.code.length+2][];
                System.arraycopy(ca.code, 0, code2, 0, z);
                code2[z+1] = ca.code[z];
                System.arraycopy(ca.code, z+1, code2, z+3,
                   ca.code.length-(z+1));
                ca.code = code2;

                // Next, insert the DUP instruction.
                ca.code[z] = new byte[1];
                ca.code[z][0] = (byte) 89;

                // Finally, insert the INVOKESTATIC instruction.
                if (exitindex == -1){
                   // This is the first time this class is encourtering the
                   // acquirelock instruction, so have to add it to the
                   // constant pool.
                   cp_info[] cpi = new cp_info[cf.constant_pool.length+6];
                   System.arraycopy(cf.constant_pool, 0, cpi, 0,
                      cf.constant_pool.length);
                   cf.constant_pool = cpi;
                   cf.constant_pool_count += 6;

                   CONSTANT_Utf8_info u1 =
                      new CONSTANT_Utf8_info("LockClient");
                   cf.constant_pool[cf.constant_pool.length-6] = u1;

                   CONSTANT_Class_info c1 = new CONSTANT_Class_info(
                      cf.constant_pool_count-6);
                   cf.constant_pool[cf.constant_pool.length-5] = c1;

                   u1 = new CONSTANT_Utf8_info("releaseLock");
                   cf.constant_pool[cf.constant_pool.length-4] = u1;

                   u1 = new CONSTANT_Utf8_info("(Ljava/lang/Object;)V");
                   cf.constant_pool[cf.constant_pool.length-3] = u1;

                   CONSTANT_NameAndType_info n1 =
                      new CONSTANT_NameAndType_info(
                      cf.constant_pool.length-4, cf.constant_pool.length-3);
                   cf.constant_pool[cf.constant_pool.length-2] = n1;

                   CONSTANT_Methodref_info m1 = new CONSTANT_Methodref_info(
                      cf.constant_pool.length-5, cf.constant_pool.length-2);
                   cf.constant_pool[cf.constant_pool.length-1] = m1;
                   exitindex = cf.constant_pool.length-1;
                }
                ca.code[z+2] = new byte[3];
                ca.code[z+2][0] = (byte) 184;
                ca.code[z+2][1] = (byte) ((exitindex >> 8) & 0xff);
                ca.code[z+2][2] = (byte) (exitindex & 0xff);


                // And lastly, increase the CODE_LENGTH and ATTRIBUTE_LENGTH
                // values.
                ca.code_length += 4;
                ca.attribute_length += 4;

                z += 1;
```

```
                }

            }

            // If we changed this method, then increase the stack size by one.
            if (changed){
                ca.max_stack++;          // Just to make sure.
            }
        }
    }


    try{

        ByteArrayOutputStream out = new ByteArrayOutputStream();
        cf.serialize(out);

        byte[] b = out.toByteArray();

        return defineClass(name, b, 0, b.length);

    }catch (Exception e){
        throw new ClassNotFoundException(name);
    }

  }

}
```

CLAIMS

1. A single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network, and
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s) whereby the corresponding replicated memory location of each said computer of said multiple system can be updated via said communicating network and all said replicated memory locations can remain substantially identical.

2. The computer as claimed in claim 1 wherein each said replicated local memory location is part of an independent local memory accessible only by the corresponding portion of said application program executing on said computer.

3. The computer as claimed in claim 1 or 2 wherein said memory location includes at least one of an asset, object or resource and has a value or content.

4. The computer as claimed in any one of claims 1 - 3 wherein a substantially identical global name is allocated to each corresponding said replicated memory location.

5. A single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network,
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s), and
initialization means which determine the initial content or value of said replicated memory location and which can be disabled.

6. The computer as claimed in claim 5 wherein said initialization means is connected to said communications port to receive an initial content or value of said replicated memory location via said communications network.

7. The computer as claimed in claim 5 or 6 wherein said initialization means is connected to said communications port to send an initial content or value of said replicated memory location via said communications network.

8. A single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least

one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network,
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s), and
finalization means which deletes said replicated memory location when all said computers no longer need to refer thereto, said finalization means being connected to said communications port to receive therefrom data transmitted over said network relating to continued reference of other computers of said multiple computer system to said replicated memory location.

9. The computer as claimed in claim 8 wherein said finalization means when said computer no longer needs to refer to said replicated memory location transfers to said communications port data indicative of the terminated reference for transmission via said communications network.

10. A single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network,
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s), and
lock acquisition and relinquishing means to respectively permit said replicated local memory location to be written to, and prevent said replicated local memory being written to, on command.

11. The computer as claimed in claim 10 wherein said command is received from said communications port.

12. The computer as claimed in claim 10 or 11 wherein said command is generated by said computer and sent to said communications port for transmission via said communications network.

13. A single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network,

updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s) whereby the corresponding replicated memory location of each said computer of said multiple system can be updated via said communicating network and all said replicated memory locations can remain substantially identical,
initialization means which determine the initial content or value of said replicated memory location and which can be disabled,
finalization means which deletes said replicated memory location when all said computers no longer need to refer thereto, said finalization means being connected to said communications port to receive therefrom data transmitted over said network relating to continued reference of other computers of said multiple computer system to said replicated memory location, and
lock acquisition and relinquishing means to respectively permit said replicated local memory location to be written to, and prevent said replicated local memory being written to, on command.

14. The computer as claimed in claim 13 wherein each said replicated local memory location is part of an independent local memory accessible only by the corresponding portion of said application program executing on said computer.

15. The computer as claimed in claim 13 or 14 wherein said memory location includes at least one of an asset, object or resource and has a value or content.

16. The computer as claimed in any one of claims 13-15 wherein a substantially identical global name is allocated to each corresponding said replicated memory location.

17. The computer as claimed in any one of claims 13-16 wherein said initialization means is connected to said communications port to receive an initial content or value of said replicated memory location via said communications network.

18. The computer as claimed in any one of claims 13-17 wherein said initialization means is connected to said communications port to send an initial content or value of said replicated memory location via said communications network.

19. The computer as claimed in any one of claims 13-18 wherein said finalization means when said computer no longer needs to refer to said replicated memory location transfers to said communications port data indicative of the terminated reference for transmission via said communications network.

20. The computer as claimed in any one of claims 13-19 wherein said command is received from said communications port.

21. The computer as claimed in any one of claims 13-20 wherein said command is generated by said computer and sent to said communications port for transmission via said communications network.

22. A multiple computer system having at least one application program each written to operate on only a single computer but running simultaneously on a plurality of computers interconnected by a communications network, wherein different portions of said application program(s) execute substantially simultaneously on different ones of said computers, wherein each computer has an independent local memory accessible only by the corresponding portion of said application program(s) and wherein for each said portion a like plurality of substantially identical objects are created, each in the corresponding computer.

23. The system as claimed in claim 22 wherein each computer has an independent local memory accessible only by the corresponding portion of said application program(s)

24. The system as claimed in claim 22 or 23 wherein each of said plurality of substantially identical objects has a substantially identical name.

25. The system as claimed in claim 24 wherein each said computer includes a distributed run time means with the distributed run time means of each said computer able to communicate with all other computers whereby if a portion of said application program(s) running on one of said computers changes the contents or value of an object in that computer then the change in content or value for said object is propagated by the distributed run time means of said one computer to all other computers to change the content or value of the corresponding object in each of said other computers.

26. The system as claimed in claim 25 wherein each said application program is modified before, during, or after loading by inserting an updating propagation routine to modify each instance at which said application program writes to memory, said updating propagation routine propagating every memory write by one computer to said other computers.

27. The system as claimed in claim 26 wherein the application program is modified in accordance with a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

28. The system as claimed in claim 26 or 27 wherein said modified application program is transferred to all said computers in accordance with a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

29. A plurality of computers interconnected via a communications link and each having an independent local memory and substantially simultaneously operating a different portion at least one application program each written to operate on only a single computer, each local memory being accessible only by the corresponding portion of said application program.

30. The plurality of computers as claimed in claim 29 wherein each said computer in operating said at least one application program reads and writes only to local memory physically located in each said computer, the contents of the local memory utilized by each said computer is fundamentally similar but not, at each instant, identical, and every one of said computers has distribution update means to distribute to all other said computers the content of any memory location updated by said one computer.

31. The plurality of computers as claimed in claim 29 or 30 wherein the local memory capacity allocated to the or each said application program is substantially identical and the total memory capacity available to the or each said application program is said allocated memory capacity.

32. The plurality of computers as claimed in claim 30 or 31 wherein all said distribution update means communicate via said communications link at a data transfer rate which is substantially less than the local memory read rate.

33. The plurality of computers as claimed in any one of claims 29-32 wherein at least some of said computers are manufactured by different manufacturers and/or have different operating systems.

34. A multiple computer system having at least one application program each written to operate on only a single computer but running substantially simultaneously on a plurality of computers interconnected by a communications network, wherein different portions of said application program(s) execute substantially simultaneously on different ones of said computers and for each said portion a like plurality of substantially identical objects are created, each in the corresponding computer and each having a substantially identical name, and wherein the initial contents of each of said identically named objects is substantially the same.

35. The system as claimed in claim 34 wherein each said computer has a local memory accessible only by the corresponding portion of said application program.

36. The system as claimed in claim 34 or 35 wherein each said computer includes a distributed run time means with the distributed run time means of each said computer able to communicate with all other computers whereby if a portion of said application program(s) running on one of said computers creates an object in that computer then the created object is propagated by the distributed run time means of said one computer to all the other computers.

37. The system as claimed in any one of claims 34-36 wherein each said application program is modified before, during, or after loading by inserting an initialization routine to modify each instance at which said application program creates an object, said initialization routine propagating every object newly created by one computer to all said other computers.

38. The system as claimed in claim 37 wherein said inserted initialization routine modifies a pre-existing initialization routine to enable the pre-existing initialization routine to execute on creation of the first of said like plurality of objects, and to disable the pre-existing initialization routine on creation of all subsequent ones of said like plurality of objects.

39. The system as claimed in claim 37 or 38 wherein the application program is modified in accordance with a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

40. The system as claimed in any one of claims 37-39 wherein said modified application program is transferred to all said computers in accordance with a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

41. A plurality of computers interconnected via a communications link and substantially simultaneously operating at least one application program each written to operation on only a single computer wherein each said computer substantially simultaneously executes a different portion of said application program(s), each said computer in operating its application program portion creates objects only in local memory physically located in each said computer, the contents of the local memory utilized by each said computer are fundamentally similar but not, at each instant, identical, and every one of said computers has distribution update means to distribute to all other said computers objects created by said one computer.

42. The plurality of computers as claimed in claim 41 wherein the local memory capacity allocated to the or each said application program is substantially identical and the total memory capacity available to the or each said application program is said allocated memory capacity.

43. The plurality of computers as claimed in claim 41 or 42 wherein all said distribution update means communicate via said communications link at a data transfer rate which is substantially less than the local memory read rate.

44. The plurality of computers as claimed in any one of claims 41-43 wherein at least some of said computers are manufactured by different manufacturers and/or have different operating systems.

45. A multiple computer system having at least one application program each written to operate only on a single computer but running substantially simultaneously on a plurality of computers interconnected by a communications network, wherein different portions of said application program(s) execute substantially simultaneously on different ones of said computers and for each said portion a like plurality of substantially identical objects are created, each in the corresponding computer and each having a substantially identical name, and wherein all said identical objects are collectively deleted when each one of said plurality of computers no longer needs to refer to their corresponding object.

46. The system as claimed in claim 45 wherein each said computer includes a distributed run time means with the distributed run time means of each said computer able to communicate with all other computers whereby if a portion of said application program(s) running on one of said computers no longer needs to refer to an object in that computer then the identity of the unreferenced object is transmitted by the distributed run time means of said one computer to a shared table or record accessible by all the other computers.

47. The system as claimed in claim 46 or 47 wherein each said application program is modified before, during, or after loading by inserting a finalization routine to modify each instance at which said application program no longer needs to refer to an object.

48. The system as claimed in claim 47 wherein said inserted finalization routine modifies a pre-existing finalization routine to enable the pre-existing finalization routine to execute if all computers no longer need to refer to their corresponding object, and to disable the pre-existing finalization routine if at least one computer does need to refer to a corresponding object.

49. The system as claimed in claim 47 or 48 wherein the application program is modified in accordance with a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

50. The system as claimed in any one of claims 47-49 wherein said modified application program is transferred to all said computers in accordance with a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

51. A plurality of computers interconnected via a communications link and operating substantially simultaneously at least one application program each written to operate only on a single computer, wherein each said computer substantially simultaneously executes a different portion of said application program(s), each said computer in operating its application program portion needs, or no longer needs to refer to an object only in local memory physically located in each said computer, the contents of the local memory utilized by each said computer is fundamentally similar but not, at each instant, identical, and every one of said computers has a finalization

routine which deletes a non-referenced object only if each one of said plurality of computers no longer needs to refer to their corresponding object.

52. The plurality of computers as claimed in claim 51 wherein the local memory capacity allocated to the or each said application program is substantially identical and the total memory capacity available to the or each said application program is said allocated memory capacity.

53. The plurality of computers as claimed in claim 51 or 52 wherein all said distribution update means communicate via said communications link at a data transfer rate which is substantially less than the local memory read rate.

54. The plurality of computers as claimed in any one of claims 51-53 wherein at least some of said computers are manufactured by different manufacturers and/or have different operating systems.

55. A multiple computer system having at least one application program each written to operate on only a single computer but running substantially simultaneously on a plurality of computers interconnected by a communications network, wherein different portions of said application program(s) execute substantially simultaneously on different ones of said computers and for each portion a like plurality of substantially identical objects are created, each in the corresponding computer and each having a substantially identical name, and said system including a lock means applicable to all said computers wherein any computer wishing to utilize a named object therein acquires an authorizing lock from said lock means which permits said utilization and which prevents all the other computers from utilizing their corresponding named object until said authorizing lock is relinquished.

56. The system as claimed in claim 55 wherein said lock means includes an acquire lock routine and a release lock routine, and both said routines are included in modifications made to said application program running on all said computers.

57. The system as claimed in claim 55 or 56 wherein said lock means further includes a shared table or record listing said named objects in use by any said computer, a lock status for each said object, and a queue of any pending lock acquisitions.

58. The system as claimed in any one of claims 55-57 wherein said lock means is located within an additional computer not running said application program and connected to said communications network.

59. The system as claimed in any one of claims 56-58 wherein each said application program is modified before, during, or after loading by inserting said acquire lock routine and said release lock routine to modify each instance at which said application program acquires and releases respectively a lock on an object.

60. The system as claimed in claim 59 wherein the application program is modified in accordance with a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

61. The system as claimed in claim 59 or 60 wherein said modified application program is transferred to all said computers in accordance with a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

62. A plurality of computers interconnected via a communications link and operating substantially simultaneously at least one application program each written to operate on only a single computer, wherein each said computer substantially

simultaneously executes a different portion of said application program(s), each said computer in operating its application program portion utilizes an object only in local memory physically located in each said computer, the contents of the local memory utilized by each said computer is fundamentally similar but not, at each instant, identical, and every one of said computers has an acquire lock routine and a release lock routine which permit utilization of the local object only by one computer and each of the remainder of said plurality of computers is locked out of utilization of their corresponding object.

63. The plurality of computers as claimed in claim 62 wherein the local memory capacity allocated to the or each said application program is substantially identical and the total memory capacity available to the or each said application program is substantially said allocated memory capacity.

64. The plurality of computers as claimed in claim 62 or 63 wherein each said computer has a distribution update means communicate via said communications link at a data transfer rate which is substantially less than the local memory read rate.

65. The plurality of computers as claimed in any one of claims 62-64 wherein at least some of said computers are manufactured by different manufacturers and/or have different operating systems.

66. A method of running simultaneously on a plurality of computers at least one application program each written to operate on only a single computer, said computers being interconnected by means of a communications network, said method comprising the step of,

(i) executing different portions of said application program(s) on different ones of said computers and for each said portion creating a like plurality of substantially identical objects each in the corresponding computer and each accessible only by the corresponding portion of said application program.

67. The method as claimed in claim 66 wherein each computer has an independent local memory which includes the corresponding identical object.

68. The method as claimed in claim 66 or 67 comprising the further step of,

(ii) naming each of said plurality of substantially identical objects with a substantially identical global name.

69. The method as claimed in any one of claims 66-68 comprising the further step of,

(iii) if a portion of said application program running on one of said computers changes the contents or value of an object in that computer, then the change in content or value of said object is propagated to all of the other computers via said communications network to change the content or value of the corresponding object in each of said other computers.

70. The method as claimed in any one of claims 66-69 including the further step of:

(iv) modifying said application program before, during or after loading by inserting an updating propagation routine to modify each instance at which said application program writes to memory, said updating propagation routine propagating every memory write by one computer to all said other computers.

71. The method as claimed in claim 70 including the further step of:

(v) modifying said application program utilizing a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to

loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

72. The method as claimed in claim 70 or 71 including the further step of:

(vi) transferring the modified application program to all said computers utilizing a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

73. A method of loading an application program written to operate only on a single computer onto each of a plurality of computers, the computers being interconnected via a communications link, and different portions of said application program(s) being substantially simultaneously executable on different computers with each computer having an independent local memory accessible only by the corresponding portion of said application program(s), the method comprising the step of modifying the application before, during, or after loading and before execution of the relevant portion of the application program.

74. The method as claimed in claim 73 wherein the modification of the application is different for different computers.

75. The method as claimed in claim 73 or 74 wherein said modifying step comprises:-

(i) detecting instructions which share memory records utilizing one of said computers,

(ii) listing all such shared memory records and providing a naming tag for each listed memory record,

(iii) detecting those instructions which write to, or manipulate the contents of, any of said listed memory records, and

(iv) generating an updating propagation routine corresponding to each said detected write or manipulate instruction, said updating propagation routine forwarding the re-written or manipulated contents and name tag of each said re-written or manipulated listed memory record to all of the others of said computers.

76. A method of operating simultaneously on a plurality of computers all interconnected via a communications link at least one application program each written to operate on only a single computer, each of said computers having at least a minimum predetermined local memory capacity, different portions of said application program(s) being substantially simultaneously executed on different ones of said computers with the local memory of each computer being only accessible by the corresponding portion of said application program(s), said method comprising the steps of:

(i) initially providing each local memory in substantially identical condition,

(ii) satisfying all memory reads and writes generated by each said application program portion from said corresponding local memory, and

(iii) communicating via said communications link all said memory writes at each said computer which take place locally to all the remainder of said plurality of computers whereby the contents of the local memory utilised by each said computer, subject to an updating data transmission delay, remains substantially identical.

77. The method as claimed in claim 76 including the further step of:

(iv) communicating said local memory writes constituting an updating data transmission at a data transfer rate which is substantially less than the local memory read rate.

78. A method of compiling or modifying an application program written to operate on only a single computer but to run simultaneously on a plurality of computers

interconnected via a communications link, with different portions of said application program(s) executing substantially simultaneously on different ones of said computers each of which has an independent local memory accessible only by the corresponding portion of said application program, said method comprising the steps of:

(i) detecting instructions which share memory records utilizing one of said computers,

(ii) listing all such shared memory records and providing a naming tag for each listed memory record,

(iii) detecting those instructions which write to, or manipulate the contents of, any of said listed memory records, and

(iv) activating an updating propagation routine following each said detected write or manipulate instruction, said updating propagation routine forwarding the re-written or manipulated contents and name tag of each said re-written or manipulated listed memory record to the remainder of said computers.

79. The method as claimed in claim 78 and carried out prior to loading the application program onto each said computer, or during loading of the application program onto each said computer, or after loading of the application program onto each said computer and before execution of the relevant portion of the application program.

80. In a multiple thread processing computer operation in which individual threads of a single application program written to operate on only a single computer are simultaneously being processed each on a different corresponding one of a plurality of computers each having an independent local memory accessible only by the corresponding thread and each being interconnected via a communications link, the improvement comprising communicating changes in the contents of local memory physically associated with the computer processing each thread to the local memory of each other said computer via said communications link.

81. The improvement as claimed in claim 80 wherein changes to the memory associated with one said thread are communicated by the computer of said one thread to all other said computers.

82. The improvement as claimed in claim 80 wherein changes to the memory associated with one said thread are transmitted to the computer associated with another said thread and are transmitted thereby to all said other computers.

83. A method of running substantially simultaneously on a plurality of computers at least one application program each written to operate on only a single computer, said computers being interconnected by means of a communications network, said method comprising the steps of:

(i) executing different portions of said application program(s) on different ones of said computers and for each said portion creating a like plurality of substantially identical objects each in the corresponding computer and each having a substantially identical name, and

(ii) creating the initial contents of each of said identically named objects substantially the same.

84. The method as claimed in claim 83 comprising the further step of,

(iii) if a portion of said application program running on one of said computers creates an object in that computer, then the created object is propagated to all of the other computers via said communications network.

85. The method as claimed in claim 83 or 84 including the further step of:

(iv) modifying said application program before, during or after loading by

inserting an initialization routine to modify each instance at which said application program creates an object, said initialization routine propagating every object created by one computer to all said other computers.

86. The method as claimed in claim 85 including the further step of:

(v) modifying said application program utilizing a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

87. The method as claimed in claim 85 or 86 including the further step of:

(vi) transferring the modified application program to all said computers utilizing a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

88. A method of compiling or modifying an application program written to operate on only a single computer to have different portions thereof to execute substantially simultaneously on different ones of a plurality of computers interconnected via a communications link, said method comprising the steps of:

(i) detecting instructions which create objects utilizing one of said computers,

(ii) activating an initialization routine following each said detected object creation instruction, said initialization routine forwarding each created object to the remainder of said computers.

89. The method as claimed in claim 88 and carried out prior to loading the application program onto each said computer, or during loading of the application program onto each said computer, or after loading of the application program onto each said computer and before execution of the relevant portion of the application program.

90. In a multiple thread processing computer operation in which individual threads of a single application program written to operate on only a single computer are substantially simultaneously being processed each on a different corresponding one of a plurality of computers interconnected via a communications link, the improvement comprising communicating objects created in local memory physically associated with the computer processing each thread to the local memory of each other said computer via said communications link.

91. The improvement as claimed in claim 90 wherein objects created in the memory associated with one said thread are communicated by the computer of said one thread to all other said computers.

92. The improvement as claimed in claim 91 wherein objects created the memory associated with one said thread are transmitted to the computer associated with another said thread and are transmitted thereby to all said other computers.

93. A method of ensuring consistent initialization of an application program written to operate on only a single computer but different portions of which are to be executed substantially simultaneously each on a different one of a plurality of computers interconnected via a communications network, said method comprising the steps of:

(i) scrutinizing said application program at, or prior to, or after loading to detect each program step defining an initialization routine, and

(ii) modifying said initialization routine to ensure consistent operation of all said computers.

94. The method as claimed in claim 93 wherein said initialization routine is modified to execute once only on the creation of a first object by any one of said

computers and is modified to be disabled on the creation of each subsequent peer copy of said object by the remainder of said computers.

95. The method claimed in claim 93 or 94 wherein step (ii) comprises the steps of:

(iii) loading and executing said initialization routine on one of said computers,

(iv) modifying said initialization routine by said one computer, and

(v) transferring said modified initialization routine to each of the remaining computers.

96. The method as claimed in claim 95 wherein said modified initialization routine is supplied by said one computer direct to each of said remaining computers.

97. The method as claimed in claim 96 wherein said modified initialization routine is supplied in cascade fashion from said one computer sequentially to each of said remaining computers.

98. The method claimed in claim 93 or 94 wherein step (ii) comprises the steps of:

(vi) loading and modifying said initialization routine on one of said computers,

(vii) said one computer sending said unmodified initialization routine to each of the remaining computers, and

(viii) each of said remaining computers modifying said initialization routine after receipt of same.

99. The method claimed in claim 98 wherein said unmodified initialization routine is supplied by said one computer directly to each of said remaining computers.

100. The method claimed in claim 98 wherein said unmodified initialization routine is supplied in cascade fashion from said one computer sequentially to each of said remaining computers.

101. A method of running substantially simultaneously on a plurality of computers at least one application program each written to operate only on a single computer, said computers being interconnected by means of a communications network, said method comprising the steps of:

(i) executing different portions of said application program(s) on different ones of said computers and for each said portion creating a like plurality of substantially identical objects each in the corresponding computer and each having a substantially identical name, and

(ii) deleting all said identical objects collectively when all of said plurality of computers no longer need to refer to their corresponding object.

102. A method as claimed in claim 101 including the further step of:

(iii) providing each said computer with a distributed run time means to communicate between said computers via said communications network.

103. A method as claimed in claim 102 including the further step of:

(iv) providing a shared table or record accessible by each said distributed run time means and in which is stored the identity of any computer which no longer requires to access an object, together with the identity of the object.

104. A method as claimed in claim 103 including the further step of:

(v) associating a counter means with said shared table, said counter means storing a count of the number of said computers which no longer require to access said object.

105. A method as claimed in claim 104 including the further step of:

(vi) providing an additional computer on which said shared program does not run and which hosts said shared table and counter, said additional computer being connected to said communications network.

106. A method of ensuring consistent finalization of an application program written to operate only on a single computer but different portions of which are to be executed substantially simultaneously each on a different one of a plurality of computers interconnected via a communications network, said method comprising the steps of:

(i) scrutinizing said application program at, or prior to, or after loading to detect each program step defining an finalization routine, and

(ii) modifying said finalization routine to ensure collective deletion of corresponding objects in all said computers only when each one of said computers no longer needs to refer to their corresponding object.

107. The method as claimed in claim 106 wherein said finalization routine is modified to execute to clean-up an object once only and on only one of said computers and when all of said computers no longer need to refer to said object.

108. The method claimed in claim 106 or 107 wherein step (ii) comprises the steps of:

(iii) loading and executing said finalization routine on one of said computers,

(iv) modifying said finalization routine by said one computer, and

(v) transferring said modified finalization routine to each of the remaining computers.

109. The method as claimed in claim 108 wherein said modified finalization routine is supplied by said one computer direct to each of said remaining computers.

110. The method as claimed in claim 108 wherein said modified finalization routine is supplied in cascade fashion from said one computer sequentially to each of said remaining computers.

111. The method claimed in claim 106 or 107 wherein step (ii) comprises the steps of:

(vi) loading and modifying said finalization routine on one of said computers,

(vii) said one computer sending said unmodified finalization routine to each of the remaining computers, and

(viii) each of said remaining computers modifying said finalization routine after receipt of same.

112. The method claimed in claim 111 wherein said unmodified finalization routine is supplied by said one computer directly to each of said remaining computers.

113. The method claimed in claim 111 wherein said unmodified finalization routine is supplied in cascade fashion from said one computer sequentially to each of said remaining computers.

114. The method as claimed in any one of claims 106-113 including the further step of:

(ix) modifying said application program utilizing a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

115. The method as claimed in claim 114 including the further step of:

(x) transferring the modified application program to all said computers utilizing a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

116. In a multiple thread processing computer operation in which individual threads of a single application program written to operate only on a single computer are substantially simultaneously being processed each on a corresponding different one of a

plurality of computers interconnected via a communications link, and in which objects in local memory physically associated with the computer processing each thread have corresponding objects in the local memory of each other said computer, the improvement comprising collectively deleting all said corresponding objects when each one of said plurality of computers no longer needs to refer to their corresponding object.

117. The improvement as claimed in claim 116 wherein an object residing in the memory associated with one said thread and to be deleted has its identity communicated by the computer of said one thread to a shared table or record accessible by all other said computers.

118. The improvement as claimed in claim 116 wherein an object residing in the memory associated with one said thread and to be deleted has its identity transmitted to the computer associated with another said thread and is transmitted thereby to a shared table or record accessible by all said other computers.

119. A method of running substantially simultaneously on a plurality of computers at least one application program each written to operate only on a single computer, said computers being interconnected by means of a communications network, said method comprising the steps of:

(i) executing different portions of said application program(s) on different ones of said computers and for each said portion creating a like plurality of substantially identical objects each in the corresponding computer and each having a substantially identical name, and

(ii) requiring any of said computers wishing to utilize a named object therein to acquire an authorizing lock which permits said utilization and which prevents all the other computers from utilizing their corresponding named object until said authorizing lock is relinquished.

120. A method as claimed in claim 119 including the further step of:

(iii) providing each said computer with a distributed run time means to communicate between said computers via said communications network.

121. A method as claimed in claim 120 including the further step of:

(iv) providing a shared table or record accessible by each said distributed run time means and in which is stored the identity of any computer which currently has to access an object, together with the identity of the object.

122. A method as claimed in claim 121 including the further step of:

(v) associating a counter means with said shared table, said counter means storing a count of the number of said computers which seek access to said object.

123. A method as claimed in claim 122 including the further step of:

(vi) providing an additional computer on which said shared program does not run and which hosts said shared table and counter, said additional computer being connected to said communications network.

124. A method of ensuring consistent synchronization of an application program written to operate only on a single computer but different portions of which are to be executed substantially simultaneously each on a different one of a plurality of computers interconnected via a communications network, said method comprising the steps of:

(i) scrutinizing said application program at, or prior to, or after loading to detect each program step defining an synchronization routine, and

(ii) modifying said synchronization routine to ensure utilization of an object by

only one computer and preventing all the remaining computers from simultaneously utilizing their corresponding objects.

125. The method claimed in claim 124 wherein step (ii) comprises the steps of:

(iii) loading and executing said synchronization routine on one of said computers,

(iv) modifying said synchronization routine by said one computer, and

(v) transferring said modified synchronization routine to each of the remaining computers.

126. The method as claimed in claim 125 wherein said modified synchronization routine is supplied by said one computer direct to each of said remaining computers.

127. The method as claimed in claim 125 wherein said modified synchronization routine is supplied in cascade fashion from said one computer sequentially to each of said remaining computers.

128. The method claimed in any one of claims 124-127 wherein step (ii) comprises the steps of:

(vi) loading and modifying said synchronization routine on one of said computers,

(vii) said one computer sending said unmodified synchronization routine to each of the remaining computers, and

(viii) each of said remaining computers modifying said synchronization routine after receipt of same.

129. The method claimed in claim 128 wherein said unmodified synchronization routine is supplied by said one computer directly to each of said remaining computers.

130. The method claimed in claim 128 wherein said unmodified synchronization routine is supplied in cascade fashion from said one computer sequentially to each of said remaining computers.

131. The method as claimed in any one of claims 124-130 including the further step of:

(ix) modifying said application program utilizing a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

132. The method as claimed in claim 131 including the further step of:

(x) transferring the modified application program to all said computers utilizing a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

133. In a multiple thread processing computer operation in which individual threads of a single application program written to operate only on a single computer are substantially simultaneously being processed each on a corresponding different one of a plurality of computers interconnected via a communications link, and in which objects in local memory physically associated with the computer processing each thread have corresponding objects in the local memory of each other said computer, the improvement comprising permitting only one of said computers to utilize an object and preventing all the remaining computers from simultaneously utilizing their corresponding object.

134. The improvement as claimed in claim 133 wherein an object residing in the memory associated with one said thread and to be utilized has its identity communicated by the computer of said one thread to a shared table or record accessible by all other said computers.

135. The improvement as claimed in claim 133 wherein an object residing in the memory associated with one said thread and to be utilized has its identity transmitted to the computer associated with another said thread and is transmitted thereby to a shared table accessible by all said other computers.

136. A computer program product comprising a set of program instructions stored in a storage medium and operable to permit at least one computer to carry out the method as claimed in any one of claims 66-135.

137. A plurality of computers interconnected via a communication network and operable to ensure consistent operation of an application program written to operate only on a single computer but running substantially simultaneously of said computers, said computers being programmed to carry out the method as claimed in any one of claims 66-135 or being loaded with the computer program product as claimed in claim 136.

138. A distributed run time adapted to enable communications between a plurality of computers, computing machines, or information appliances.

139. A modifier for modifying an application program written to execute on a single computer whereby the modified application program executes substantially simultaneously on a plurality of network computers or computing machines.

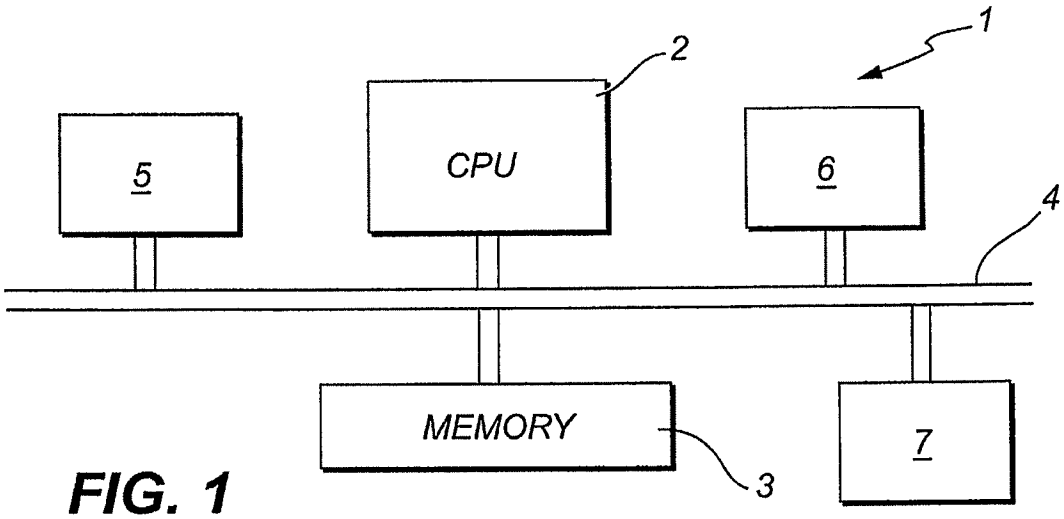

FIG. 1
PRIOR ART

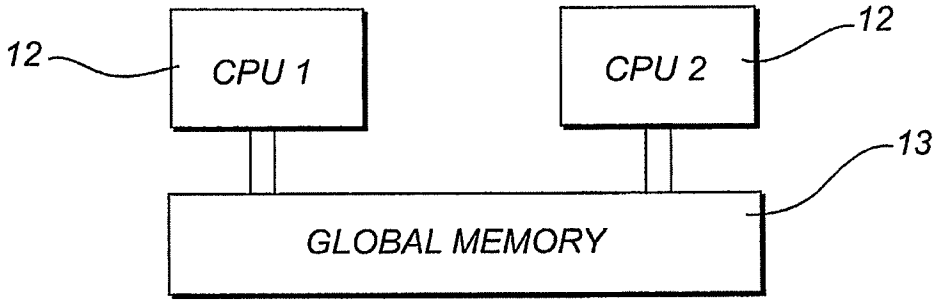

FIG. 2
PRIOR ART

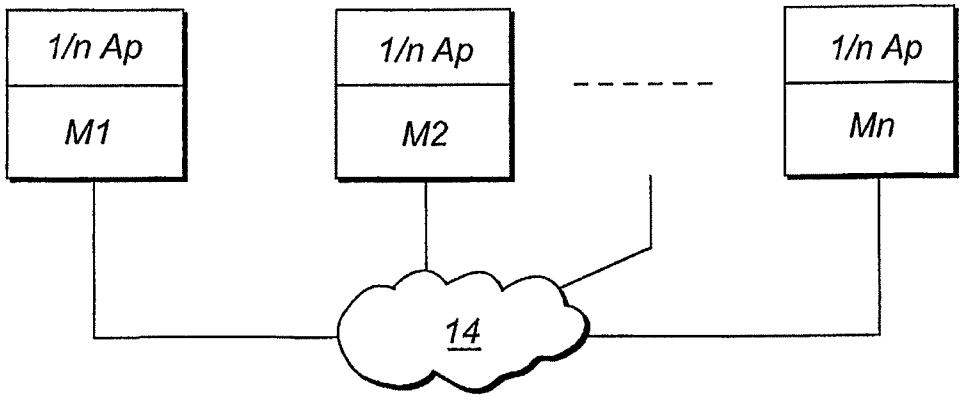

FIG. 3
PRIOR ART

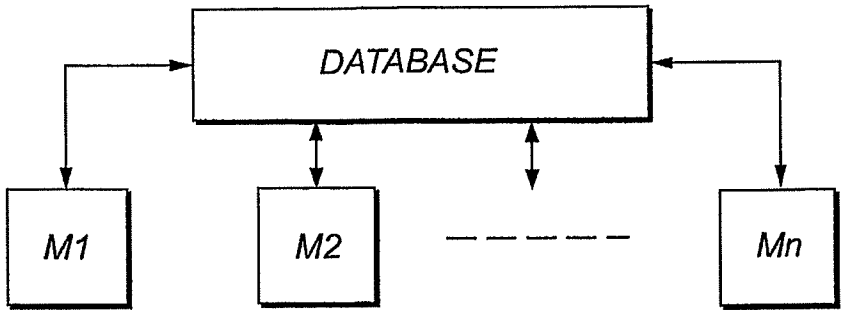

FIG. 4
PRIOR ART

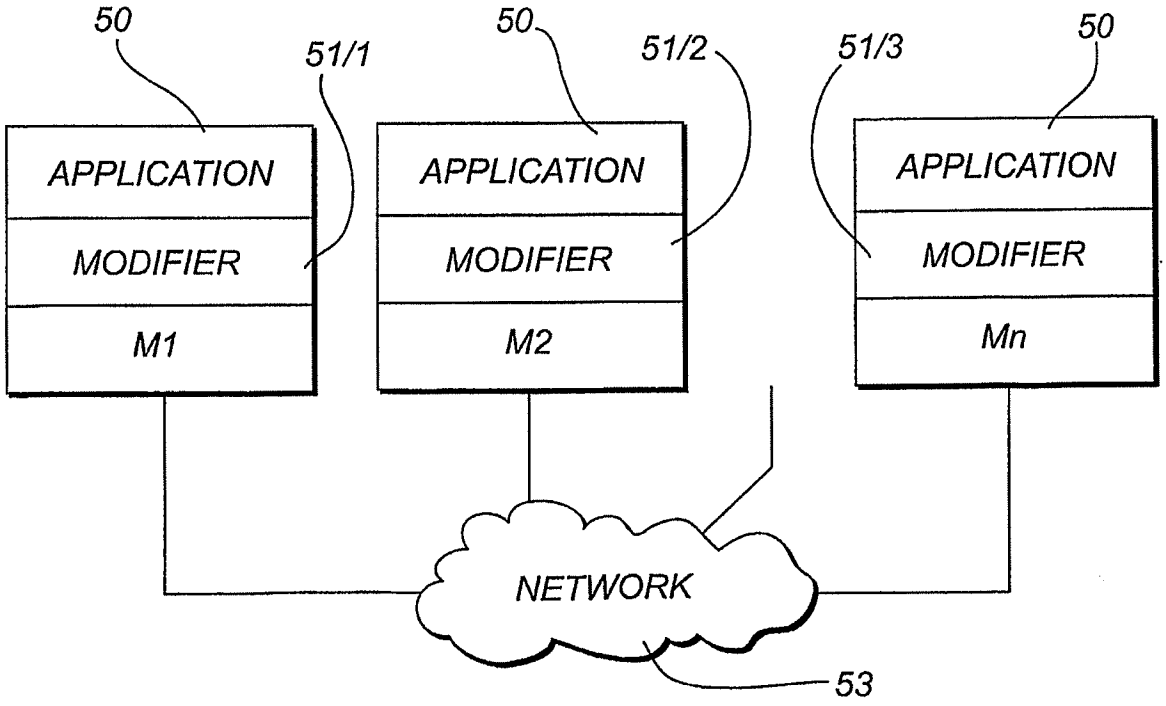

FIG. 5

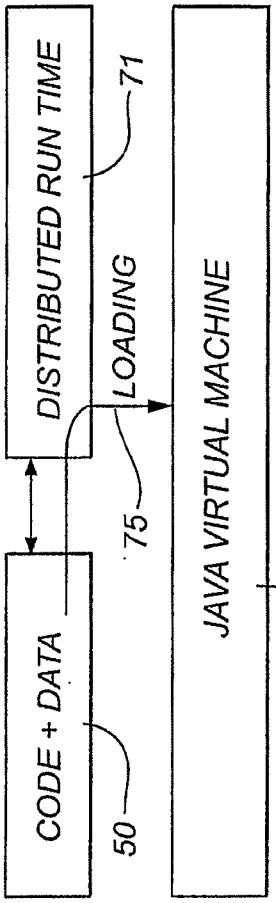

FIG. 7

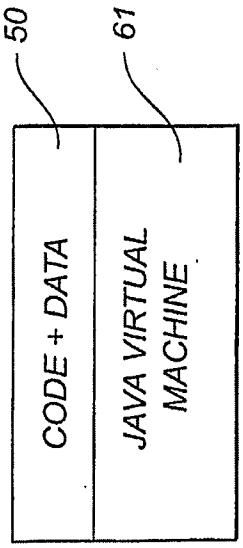

FIG. 6
PRIOR ART

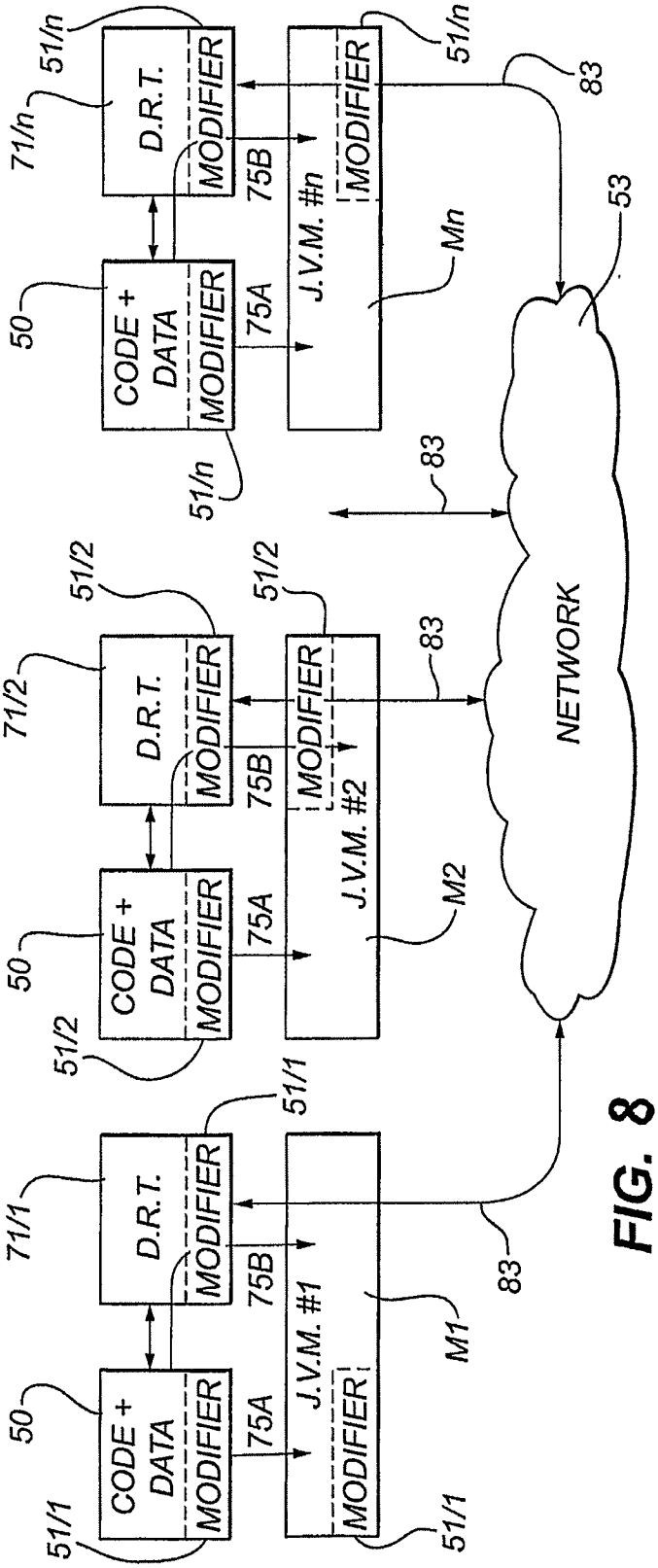

FIG. 8

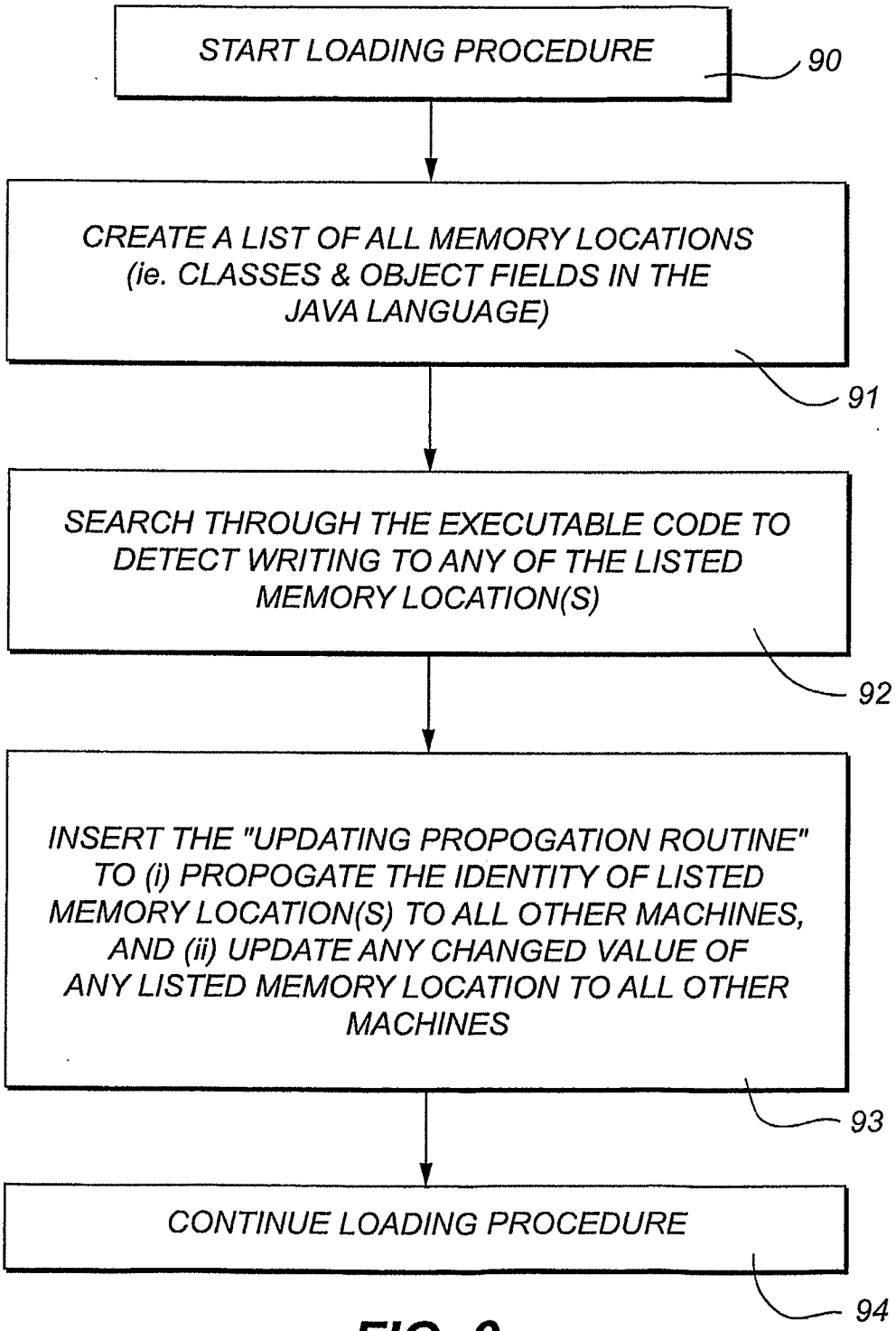
START LOADING PROCEDURE
90
CREATE A LIST OF ALL MEMORY LOCATIONS (ie. CLASSES & OBJECT FIELDS IN THE JAVA LANGUAGE)
91
SEARCH THROUGH THE EXECUTABLE CODE TO DETECT WRITING TO ANY OF THE LISTED MEMORY LOCATION(S)
92
INSERT THE "UPDATING PROPOGATION ROUTINE" TO (i) PROPOGATE THE IDENTITY OF LISTED MEMORY LOCATION(S) TO ALL OTHER MACHINES, AND (ii) UPDATE ANY CHANGED VALUE OF ANY LISTED MEMORY LOCATION TO ALL OTHER MACHINES
93
CONTINUE LOADING PROCEDURE
94

FIG. 9

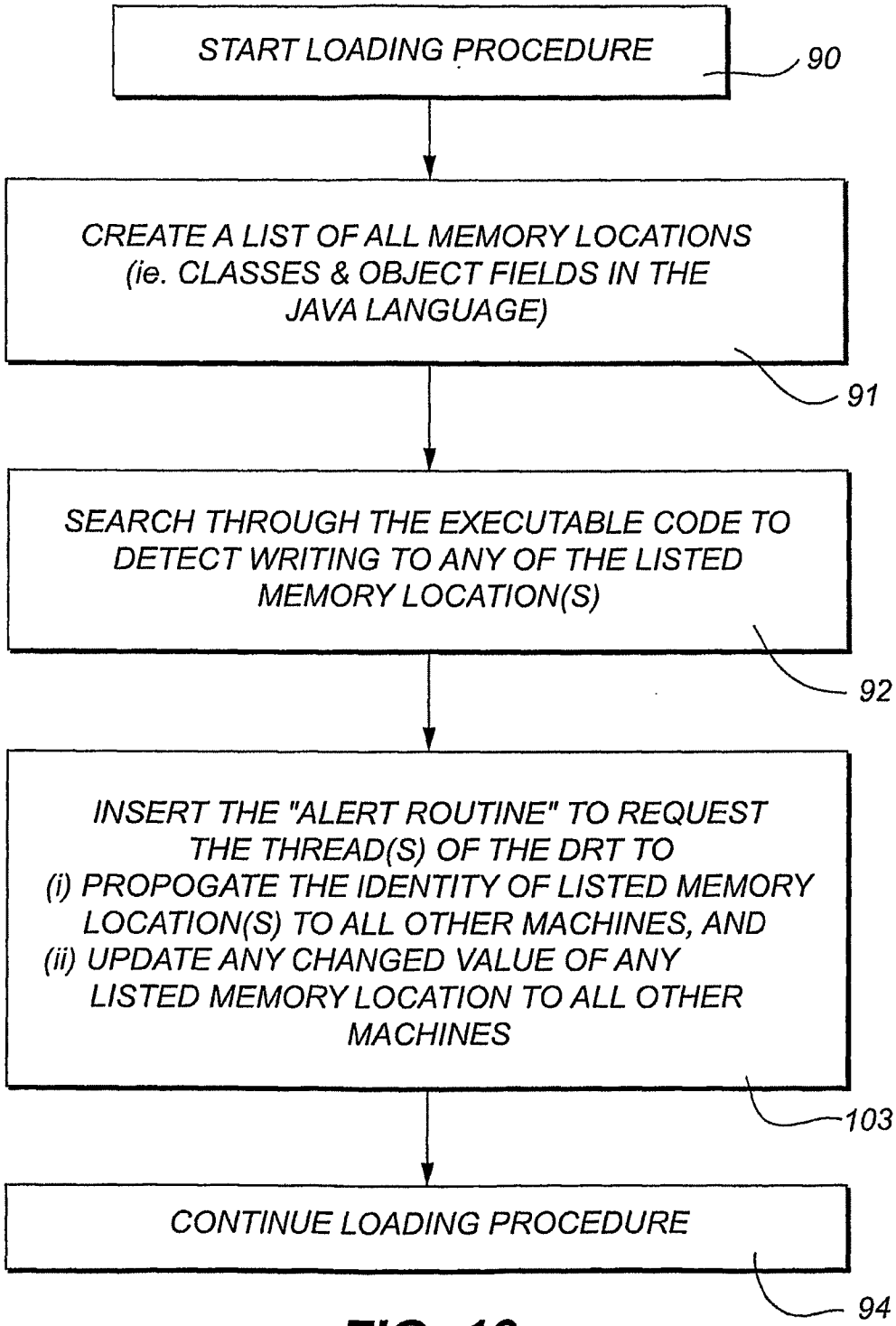
START LOADING PROCEDURE
90
CREATE A LIST OF ALL MEMORY LOCATIONS (ie. CLASSES & OBJECT FIELDS IN THE JAVA LANGUAGE)
91
SEARCH THROUGH THE EXECUTABLE CODE TO DETECT WRITING TO ANY OF THE LISTED MEMORY LOCATION(S)
92
INSERT THE "ALERT ROUTINE" TO REQUEST THE THREAD(S) OF THE DRT TO
(i) PROPOGATE THE IDENTITY OF LISTED MEMORY LOCATION(S) TO ALL OTHER MACHINES, AND
(ii) UPDATE ANY CHANGED VALUE OF ANY LISTED MEMORY LOCATION TO ALL OTHER MACHINES
103
CONTINUE LOADING PROCEDURE
94

FIG. 10

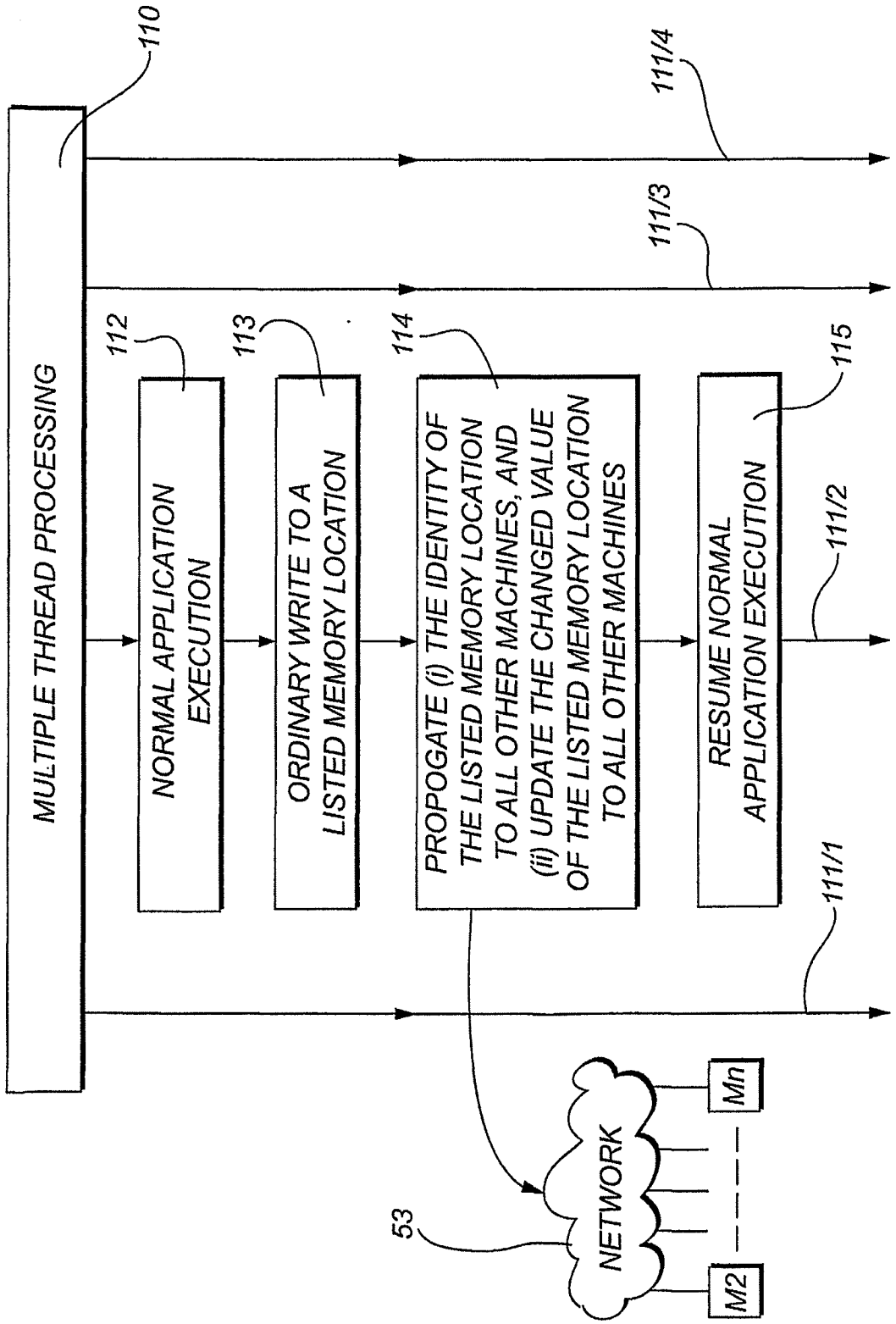
MULTIPLE THREAD PROCESSING
110
NORMAL APPLICATION EXECUTION
112
ORDINARY WRITE TO A LISTED MEMORY LOCATION
113
PROPOGATE (i) THE IDENTITY OF THE LISTED MEMORY LOCATION TO ALL OTHER MACHINES, AND (ii) UPDATE THE CHANGED VALUE OF THE LISTED MEMORY LOCATION TO ALL OTHER MACHINES
114
RESUME NORMAL APPLICATION EXECUTION
115
111/1
111/2
111/3
111/4
NETWORK
53
M2
Mn

FIG.11

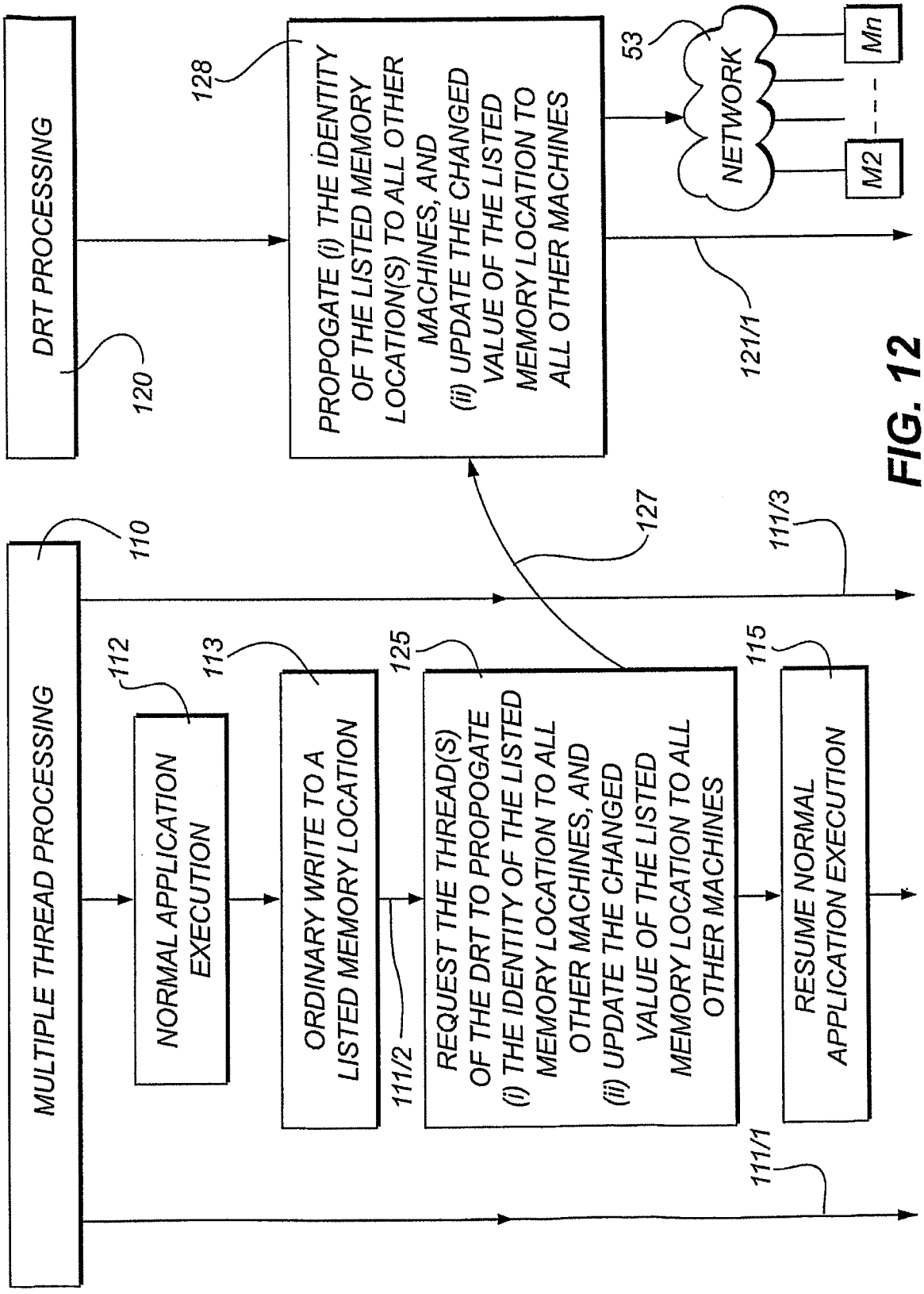
MULTIPLE THREAD PROCESSING
110
NORMAL APPLICATION EXECUTION
112
ORDINARY WRITE TO A LISTED MEMORY LOCATION
113
111/2
REQUEST THE THREAD(S) OF THE DRT TO PROPOGATE (i) THE IDENTITY OF THE LISTED MEMORY LOCATION TO ALL OTHER MACHINES, AND (ii) UPDATE THE CHANGED VALUE OF THE LISTED MEMORY LOCATION TO ALL OTHER MACHINES
125
127
RESUME NORMAL APPLICATION EXECUTION
115
111/1
111/3
DRT PROCESSING
120
PROPOGATE (i) THE IDENTITY OF THE LISTED MEMORY LOCATION(S) TO ALL OTHER MACHINES, AND (ii) UPDATE THE CHANGED VALUE OF THE LISTED MEMORY LOCATION TO ALL OTHER MACHINES
128
121/1
53
NETWORK
M2
Mn

FIG. 12

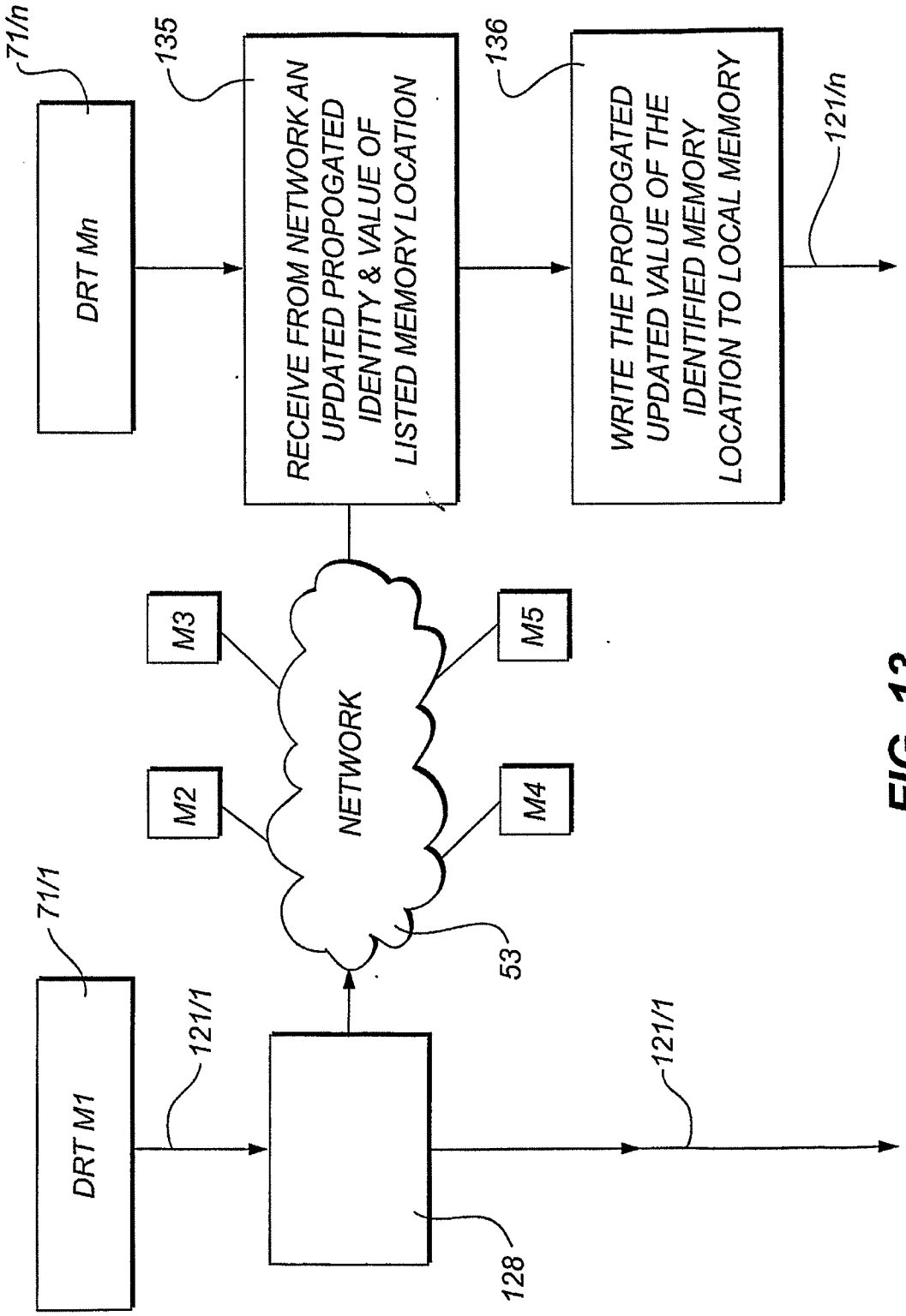
71/n
DRT Mn
135
RECEIVE FROM NETWORK AN UPDATED PROPOGATED IDENTITY & VALUE OF LISTED MEMORY LOCATION
136
WRITE THE PROPOGATED UPDATED VALUE OF THE IDENTIFIED MEMORY LOCATION TO LOCAL MEMORY
121/n
M3
M5
NETWORK
M2
M4
53
71/1
DRT M1
121/1
121/1
128

FIG. 13

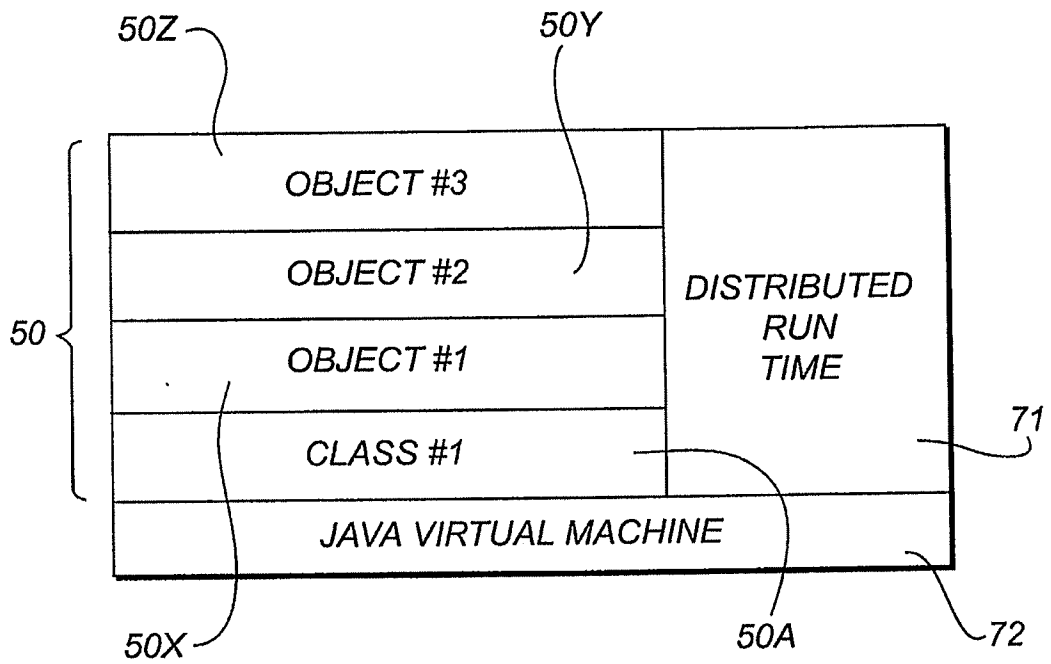

FIG. 14
PRIOR ART

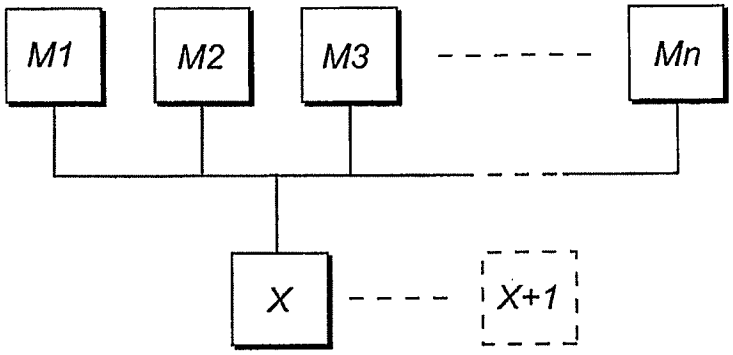
M1
M2
M3
Mn
X
X+1

FIG. 15

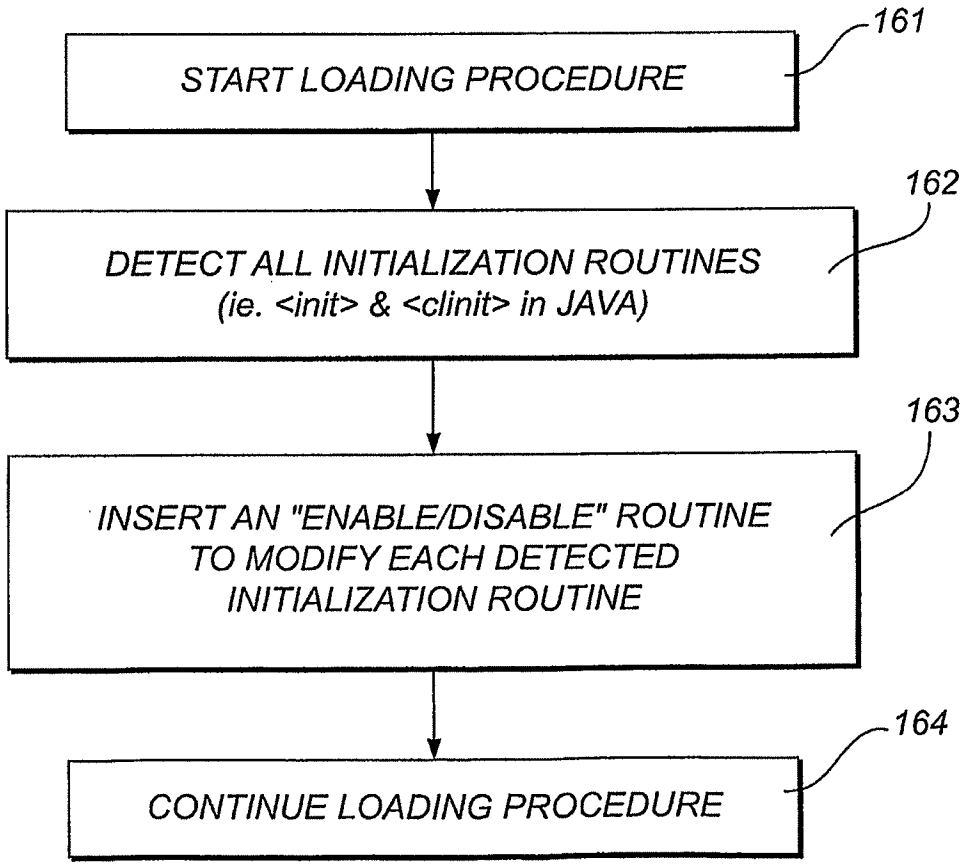
161
START LOADING PROCEDURE
162
DETECT ALL INITIALIZATION ROUTINES
(ie. <init> & <clinit> in JAVA)
163
INSERT AN "ENABLE/DISABLE" ROUTINE
TO MODIFY EACH DETECTED
INITIALIZATION ROUTINE
164
CONTINUE LOADING PROCEDURE

FIG. 16

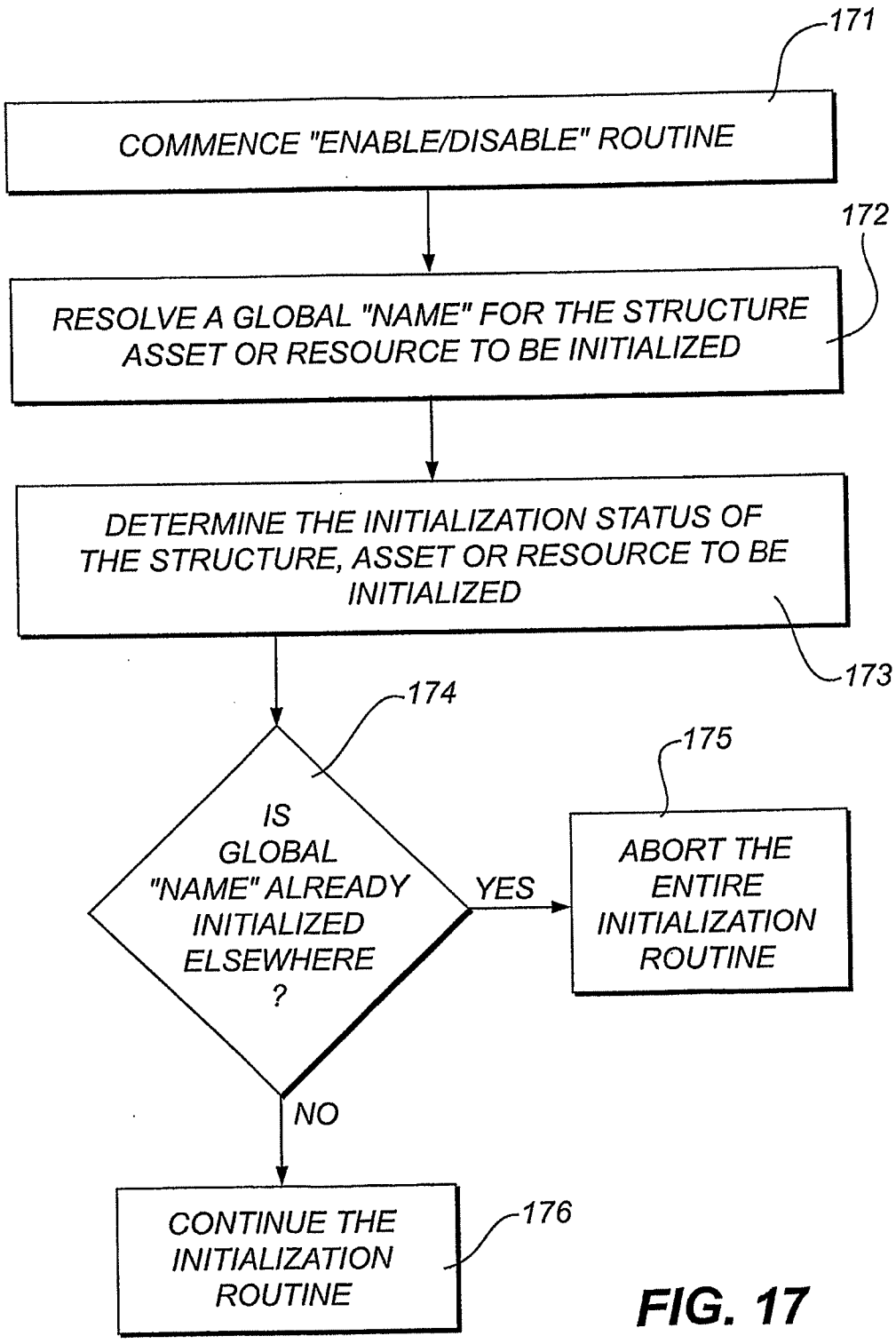
171
COMMENCE "ENABLE/DISABLE" ROUTINE
172
RESOLVE A GLOBAL "NAME" FOR THE STRUCTURE ASSET OR RESOURCE TO BE INITIALIZED
DETERMINE THE INITIALIZATION STATUS OF THE STRUCTURE, ASSET OR RESOURCE TO BE INITIALIZED
173
174
IS GLOBAL "NAME" ALREADY INITIALIZED ELSEWHERE ?
YES
175
ABORT THE ENTIRE INITIALIZATION ROUTINE
NO
CONTINUE THE INITIALIZATION ROUTINE
176

FIG. 17

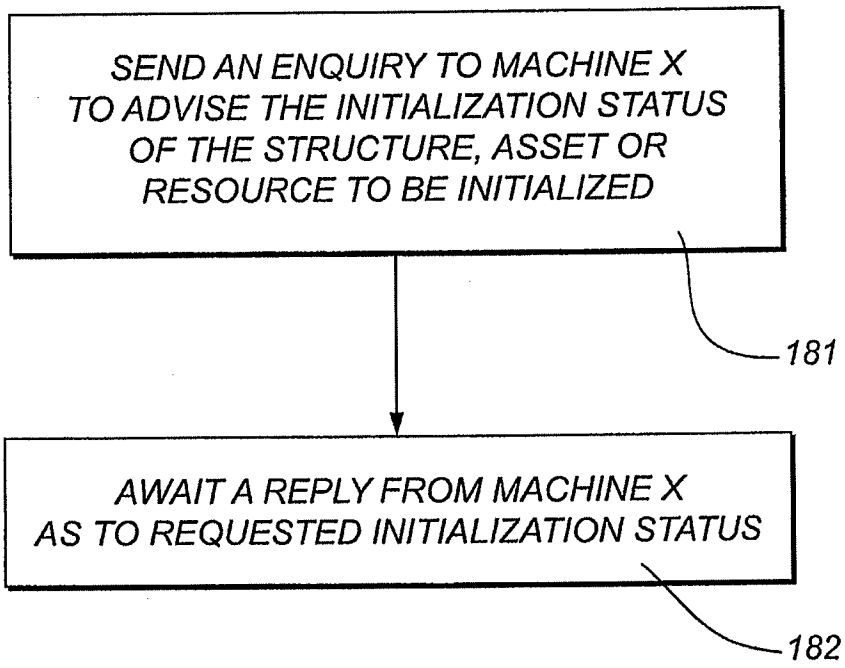
SEND AN ENQUIRY TO MACHINE X TO ADVISE THE INITIALIZATION STATUS OF THE STRUCTURE, ASSET OR RESOURCE TO BE INITIALIZED
181
AWAIT A REPLY FROM MACHINE X AS TO REQUESTED INITIALIZATION STATUS
182

FIG. 18

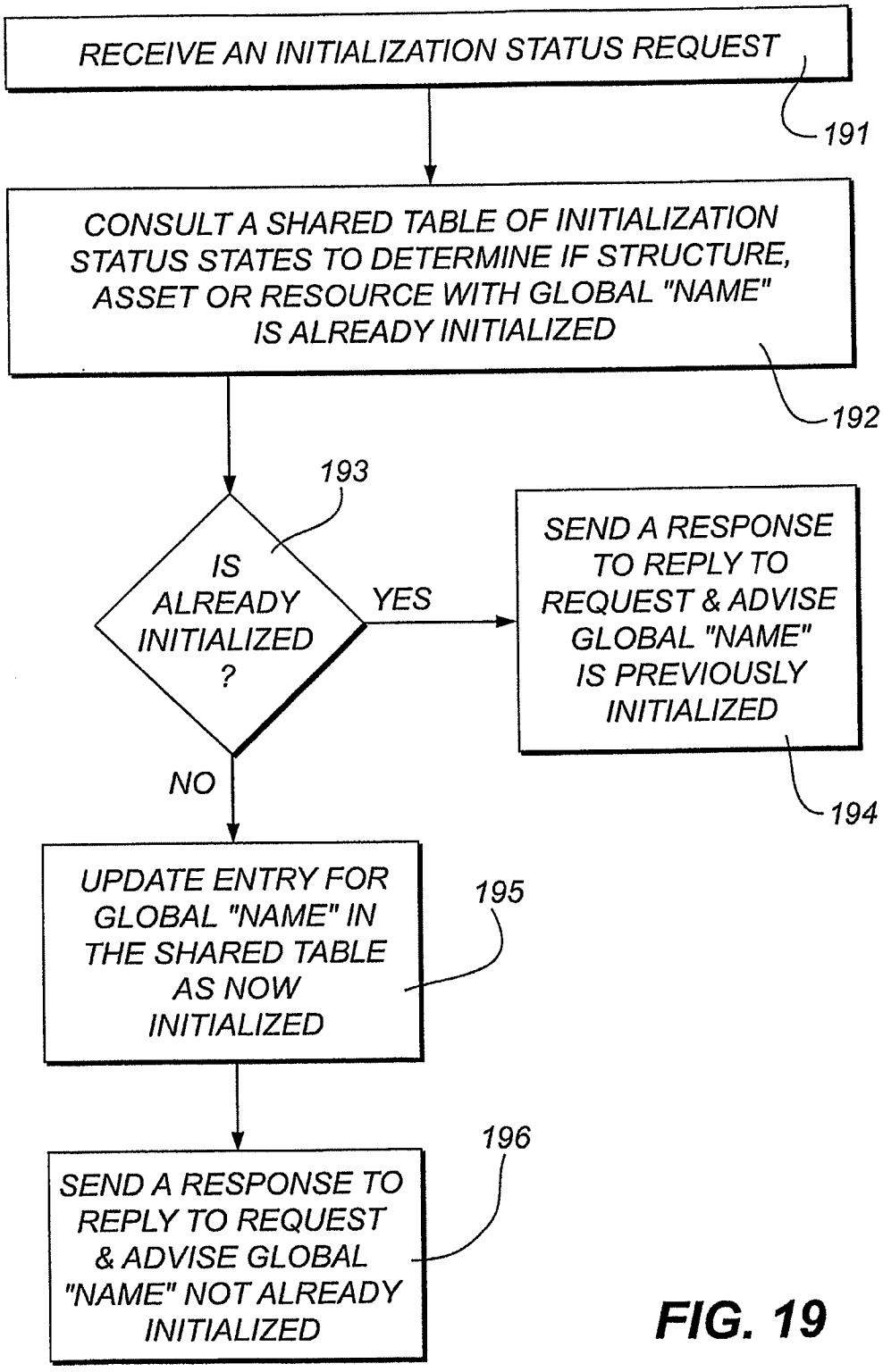
RECEIVE AN INITIALIZATION STATUS REQUEST
191
CONSULT A SHARED TABLE OF INITIALIZATION STATUS STATES TO DETERMINE IF STRUCTURE, ASSET OR RESOURCE WITH GLOBAL "NAME" IS ALREADY INITIALIZED
192
193
IS ALREADY INITIALIZED ?
YES
SEND A RESPONSE TO REPLY TO REQUEST & ADVISE GLOBAL "NAME" IS PREVIOUSLY INITIALIZED
194
NO
UPDATE ENTRY FOR GLOBAL "NAME" IN THE SHARED TABLE AS NOW INITIALIZED
195
SEND A RESPONSE TO REPLY TO REQUEST & ADVISE GLOBAL "NAME" NOT ALREADY INITIALIZED
196

FIG. 19

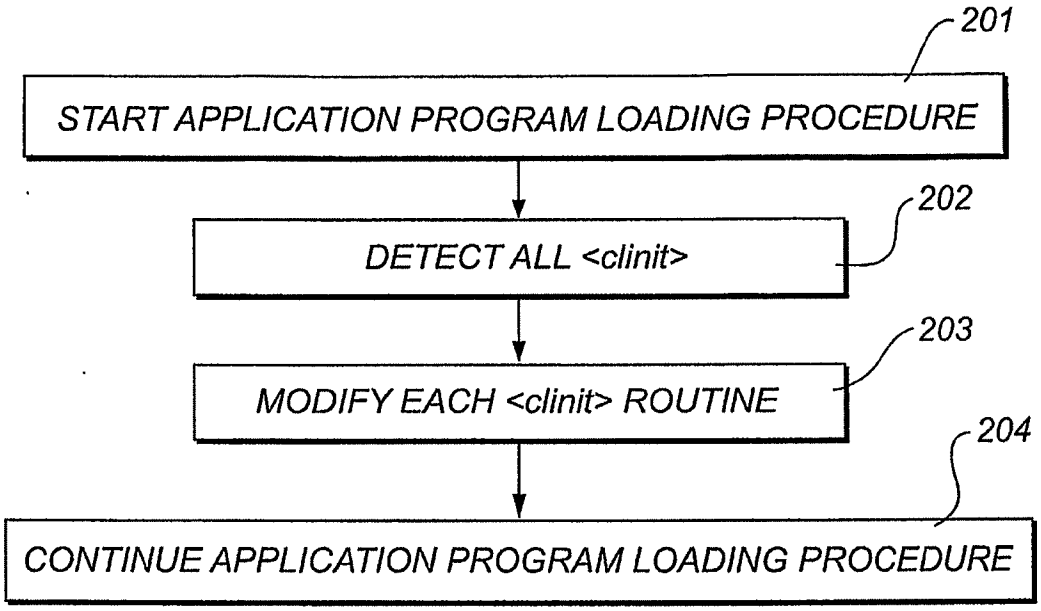
201
START APPLICATION PROGRAM LOADING PROCEDURE
202
DETECT ALL <clinit>
203
MODIFY EACH <clinit> ROUTINE
204
CONTINUE APPLICATION PROGRAM LOADING PROCEDURE

FIG. 20

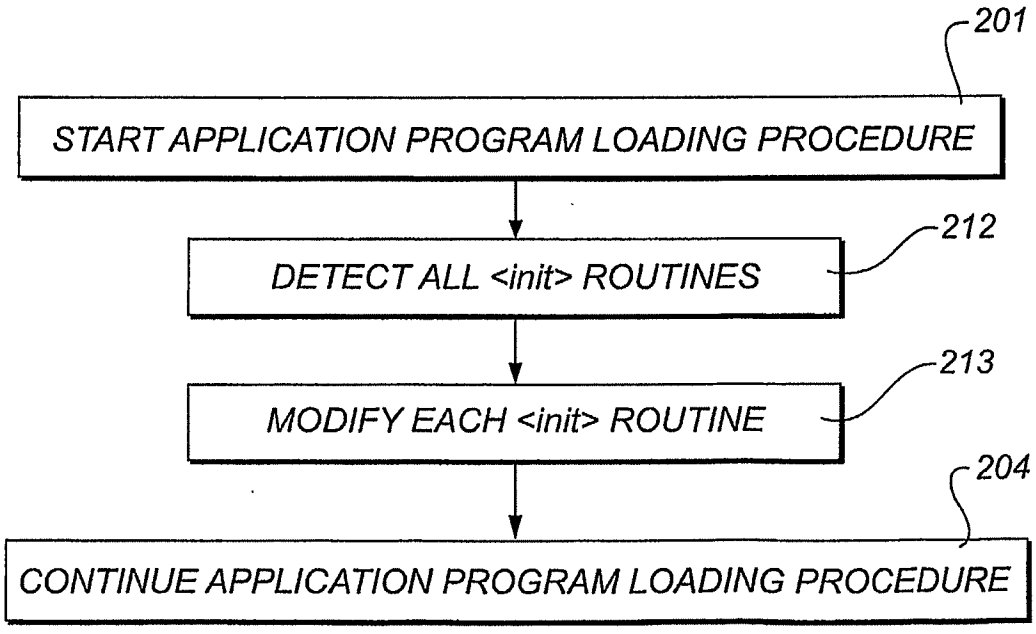
201
START APPLICATION PROGRAM LOADING PROCEDURE
212
DETECT ALL <init> ROUTINES
213
MODIFY EACH <init> ROUTINE
204
CONTINUE APPLICATION PROGRAM LOADING PROCEDURE

FIG. 21

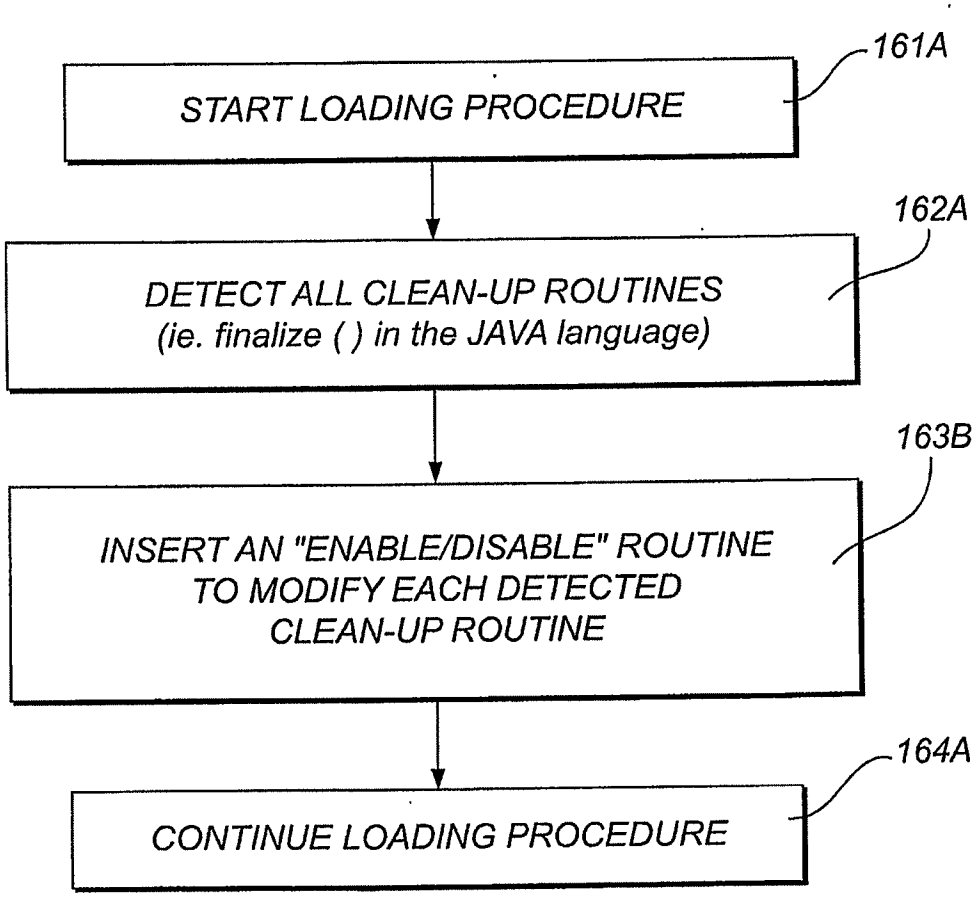
161A
START LOADING PROCEDURE
162A
DETECT ALL CLEAN-UP ROUTINES
(ie. finalize ( ) in the JAVA language)
163B
INSERT AN "ENABLE/DISABLE" ROUTINE
TO MODIFY EACH DETECTED
CLEAN-UP ROUTINE
164A
CONTINUE LOADING PROCEDURE

FIG. 22

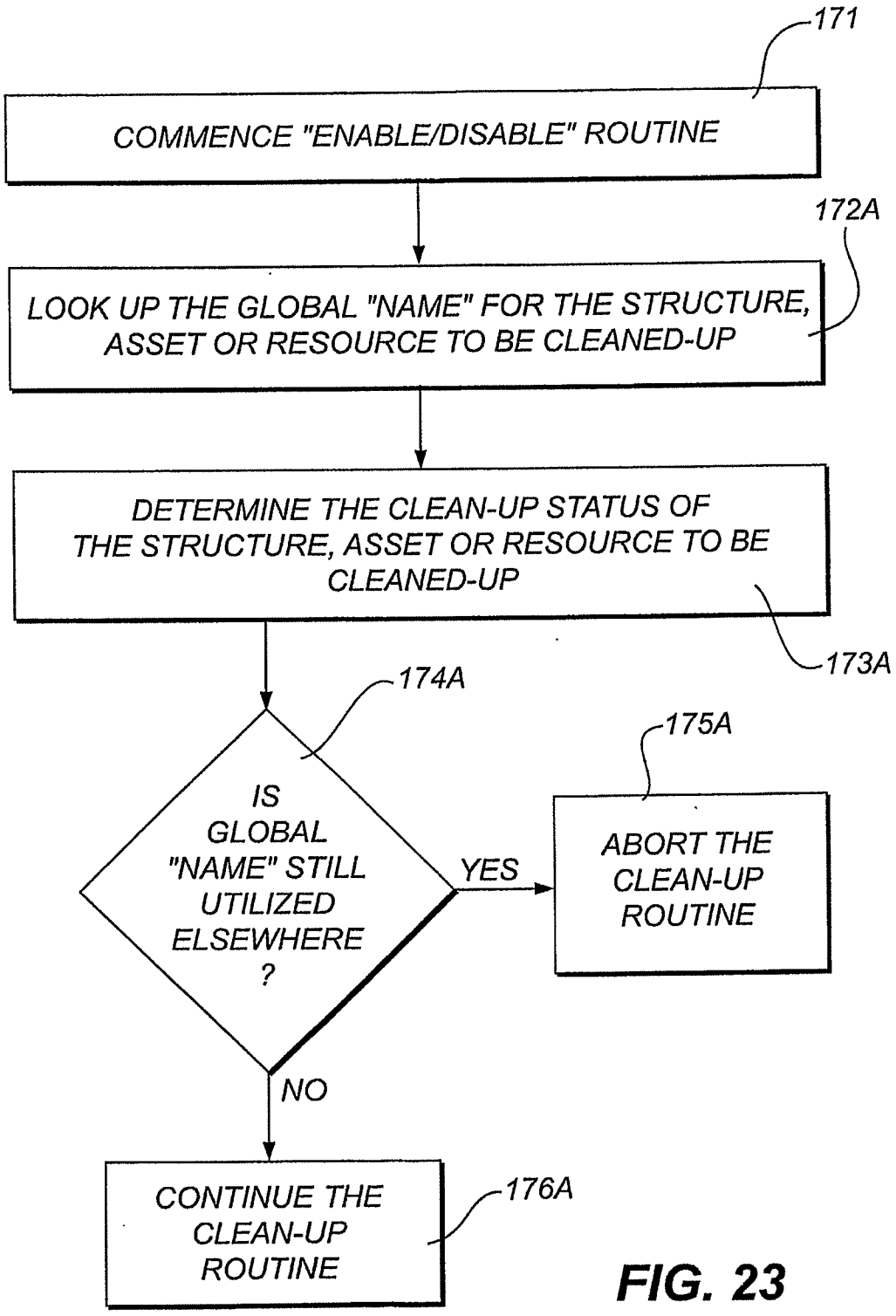
171
COMMENCE "ENABLE/DISABLE" ROUTINE
172A
LOOK UP THE GLOBAL "NAME" FOR THE STRUCTURE, ASSET OR RESOURCE TO BE CLEANED-UP
DETERMINE THE CLEAN-UP STATUS OF THE STRUCTURE, ASSET OR RESOURCE TO BE CLEANED-UP
173A
174A
IS GLOBAL "NAME" STILL UTILIZED ELSEWHERE ?
YES
175A
ABORT THE CLEAN-UP ROUTINE
NO
CONTINUE THE CLEAN-UP ROUTINE
176A

FIG. 23

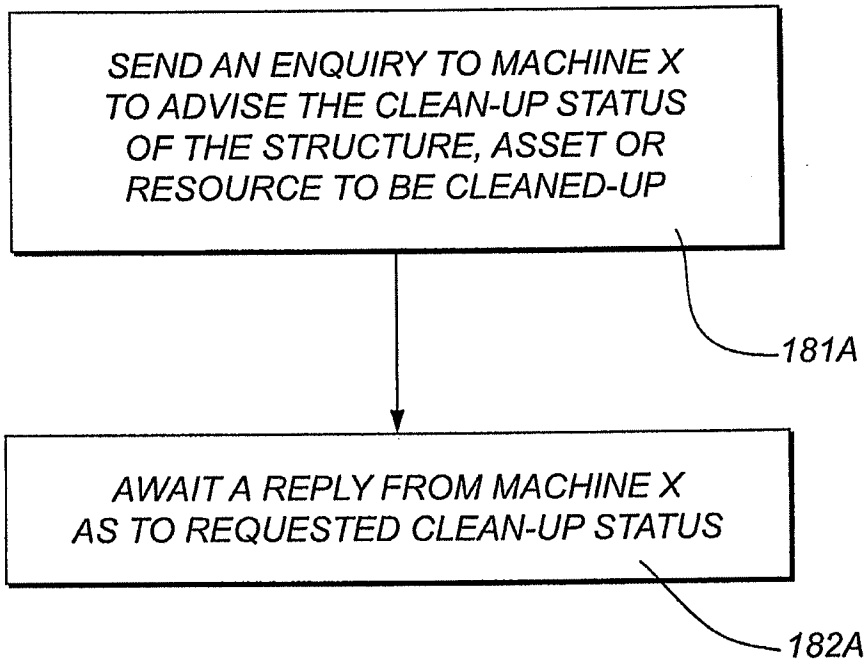
SEND AN ENQUIRY TO MACHINE X TO ADVISE THE CLEAN-UP STATUS OF THE STRUCTURE, ASSET OR RESOURCE TO BE CLEANED-UP
181A
AWAIT A REPLY FROM MACHINE X AS TO REQUESTED CLEAN-UP STATUS
182A

FIG. 24

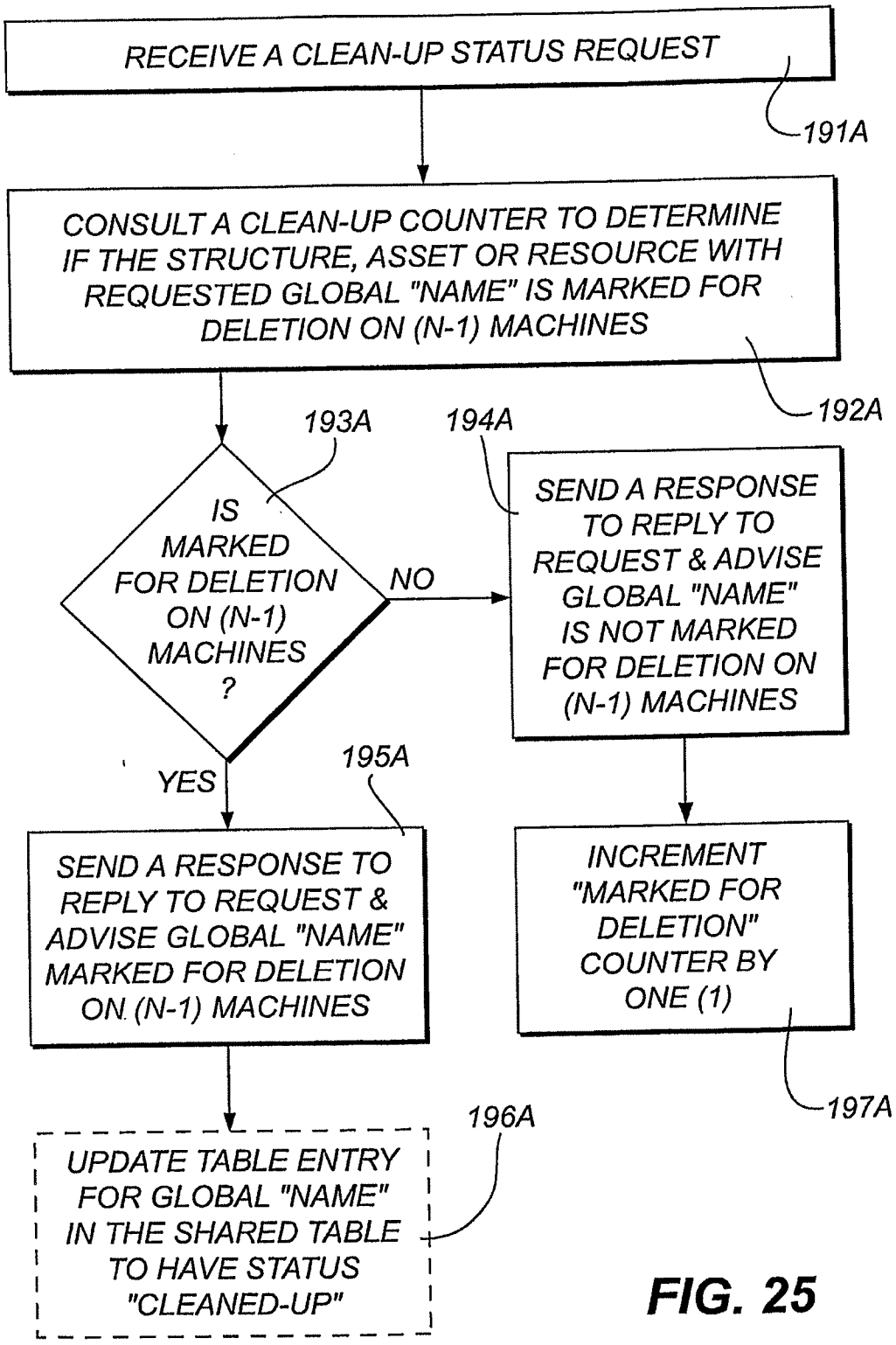
RECEIVE A CLEAN-UP STATUS REQUEST
191A
CONSULT A CLEAN-UP COUNTER TO DETERMINE IF THE STRUCTURE, ASSET OR RESOURCE WITH REQUESTED GLOBAL "NAME" IS MARKED FOR DELETION ON (N-1) MACHINES
192A
193A
IS MARKED FOR DELETION ON (N-1) MACHINES ?
NO
194A
SEND A RESPONSE TO REPLY TO REQUEST & ADVISE GLOBAL "NAME" IS NOT MARKED FOR DELETION ON (N-1) MACHINES
YES
195A
SEND A RESPONSE TO REPLY TO REQUEST & ADVISE GLOBAL "NAME" MARKED FOR DELETION ON (N-1) MACHINES
INCREMENT "MARKED FOR DELETION" COUNTER BY ONE (1)
197A
196A
UPDATE TABLE ENTRY FOR GLOBAL "NAME" IN THE SHARED TABLE TO HAVE STATUS "CLEANED-UP"

FIG. 25

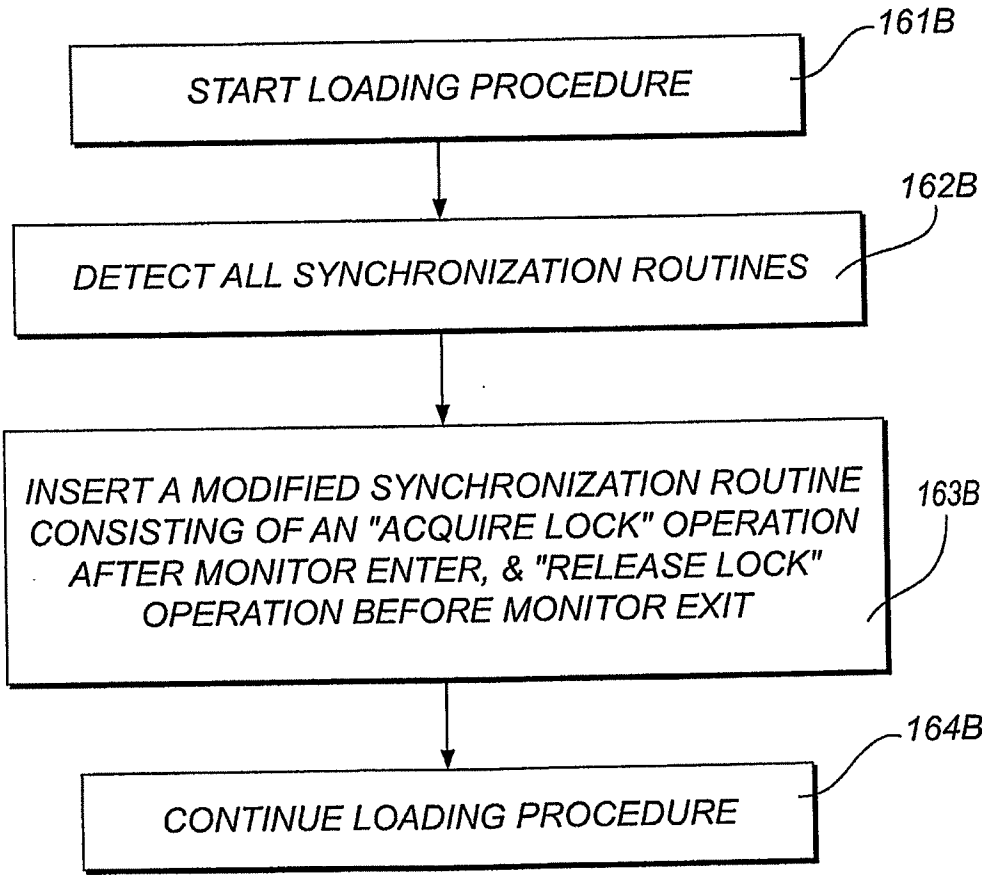
161B
START LOADING PROCEDURE
162B
DETECT ALL SYNCHRONIZATION ROUTINES
INSERT A MODIFIED SYNCHRONIZATION ROUTINE CONSISTING OF AN "ACQUIRE LOCK" OPERATION AFTER MONITOR ENTER, & "RELEASE LOCK" OPERATION BEFORE MONITOR EXIT
163B
164B
CONTINUE LOADING PROCEDURE

FIG. 26

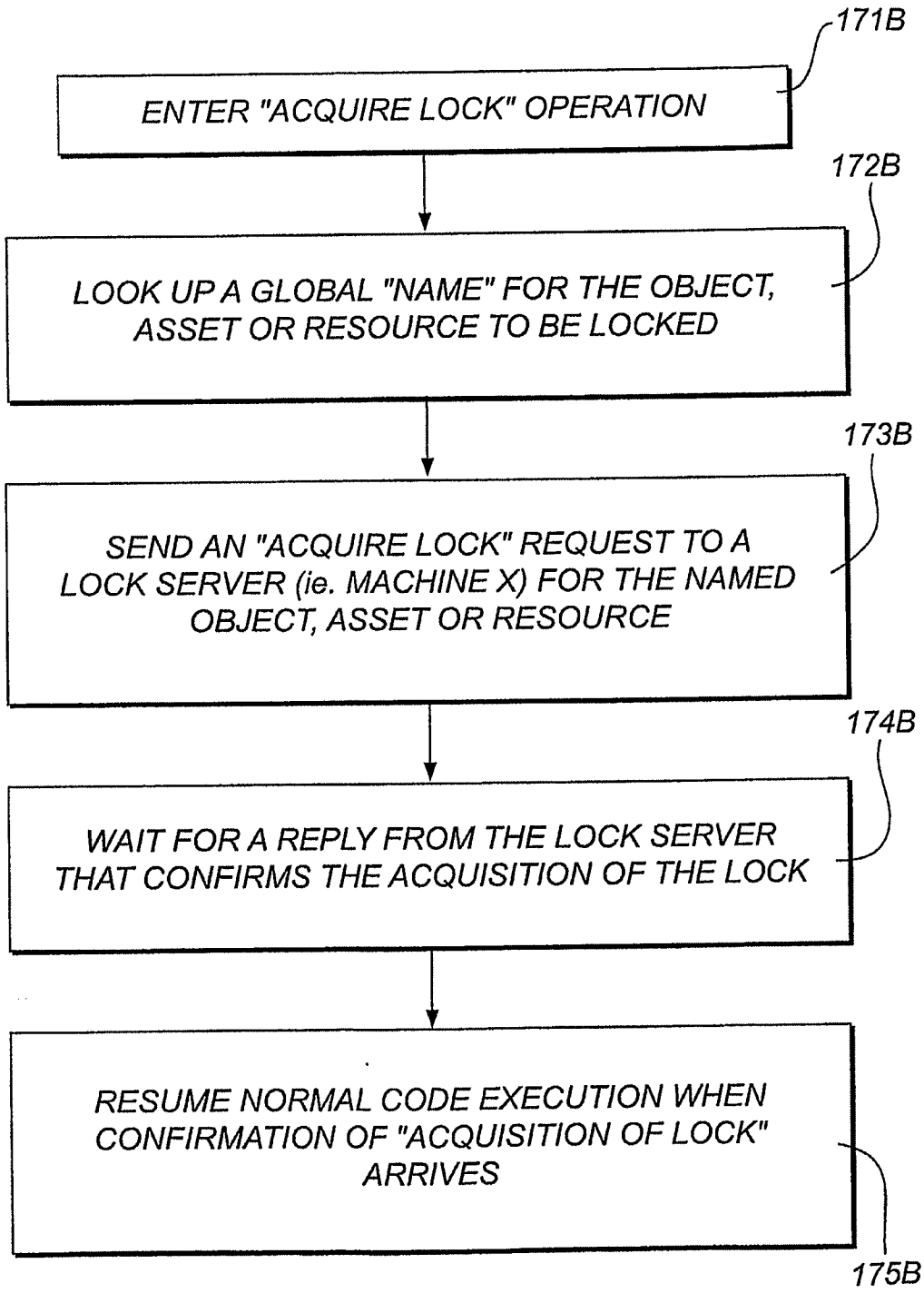
171B
ENTER "ACQUIRE LOCK" OPERATION
172B
LOOK UP A GLOBAL "NAME" FOR THE OBJECT, ASSET OR RESOURCE TO BE LOCKED
173B
SEND AN "ACQUIRE LOCK" REQUEST TO A LOCK SERVER (ie. MACHINE X) FOR THE NAMED OBJECT, ASSET OR RESOURCE
174B
WAIT FOR A REPLY FROM THE LOCK SERVER THAT CONFIRMS THE ACQUISITION OF THE LOCK
RESUME NORMAL CODE EXECUTION WHEN CONFIRMATION OF "ACQUISITION OF LOCK" ARRIVES
175B

FIG. 27

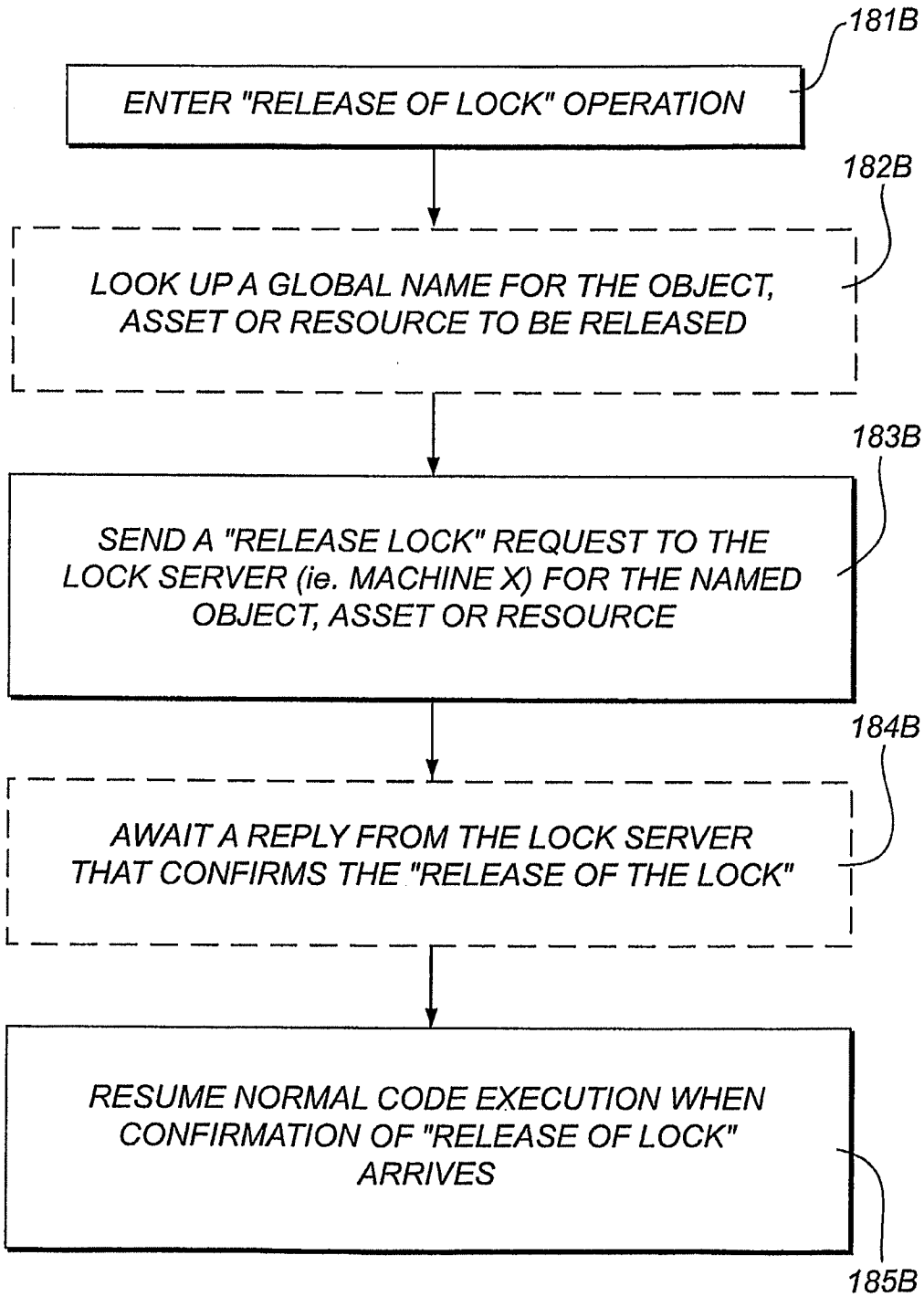
181B
ENTER "RELEASE OF LOCK" OPERATION
182B
LOOK UP A GLOBAL NAME FOR THE OBJECT, ASSET OR RESOURCE TO BE RELEASED
183B
SEND A "RELEASE LOCK" REQUEST TO THE LOCK SERVER (ie. MACHINE X) FOR THE NAMED OBJECT, ASSET OR RESOURCE
184B
AWAIT A REPLY FROM THE LOCK SERVER THAT CONFIRMS THE "RELEASE OF THE LOCK"
RESUME NORMAL CODE EXECUTION WHEN CONFIRMATION OF "RELEASE OF LOCK" ARRIVES
185B

FIG. 28

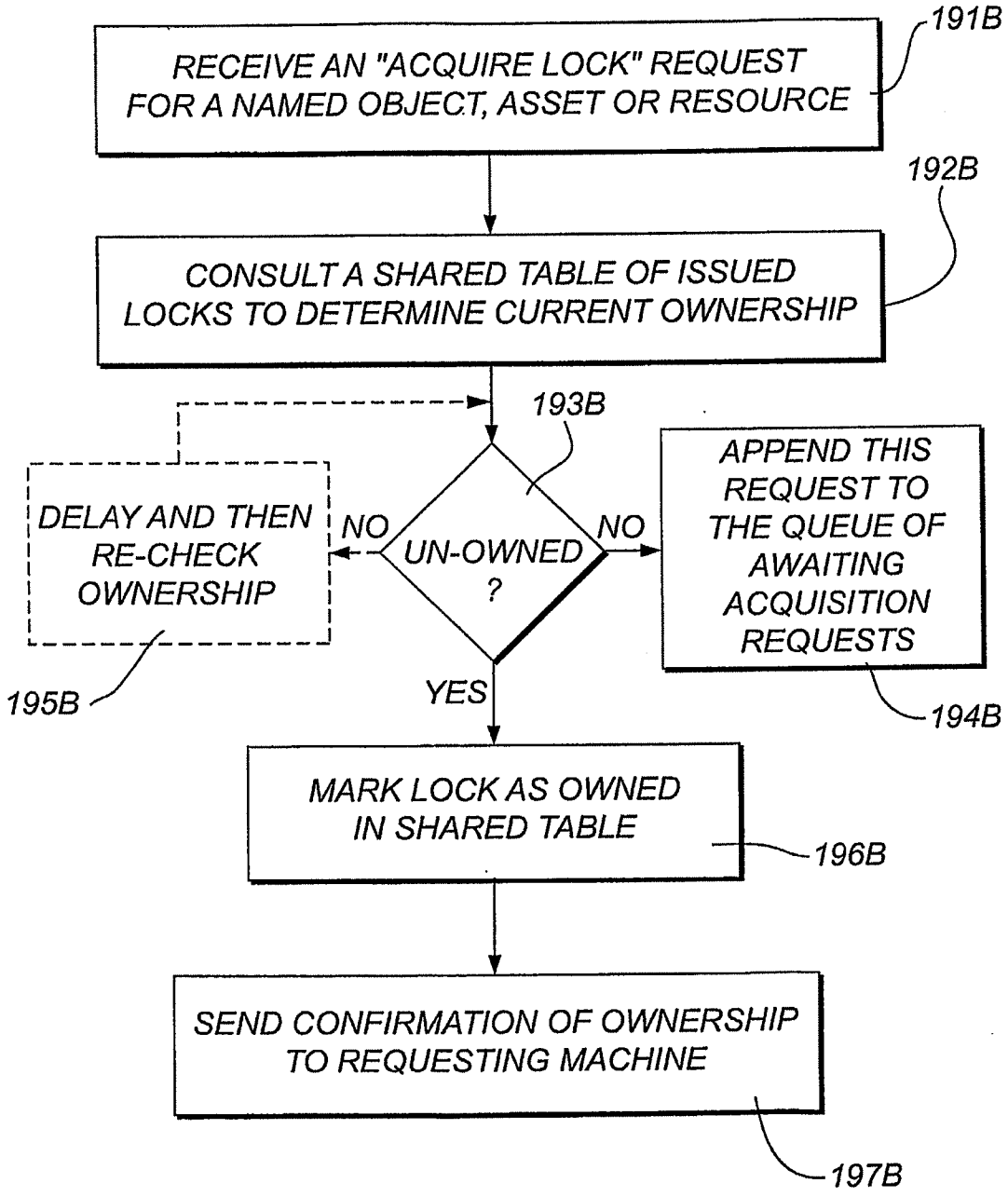
RECEIVE AN "ACQUIRE LOCK" REQUEST FOR A NAMED OBJECT, ASSET OR RESOURCE
191B
CONSULT A SHARED TABLE OF ISSUED LOCKS TO DETERMINE CURRENT OWNERSHIP
192B
193B
UN-OWNED ?
NO
NO
YES
DELAY AND THEN RE-CHECK OWNERSHIP
195B
APPEND THIS REQUEST TO THE QUEUE OF AWAITING ACQUISITION REQUESTS
194B
MARK LOCK AS OWNED IN SHARED TABLE
196B
SEND CONFIRMATION OF OWNERSHIP TO REQUESTING MACHINE
197B

FIG. 29

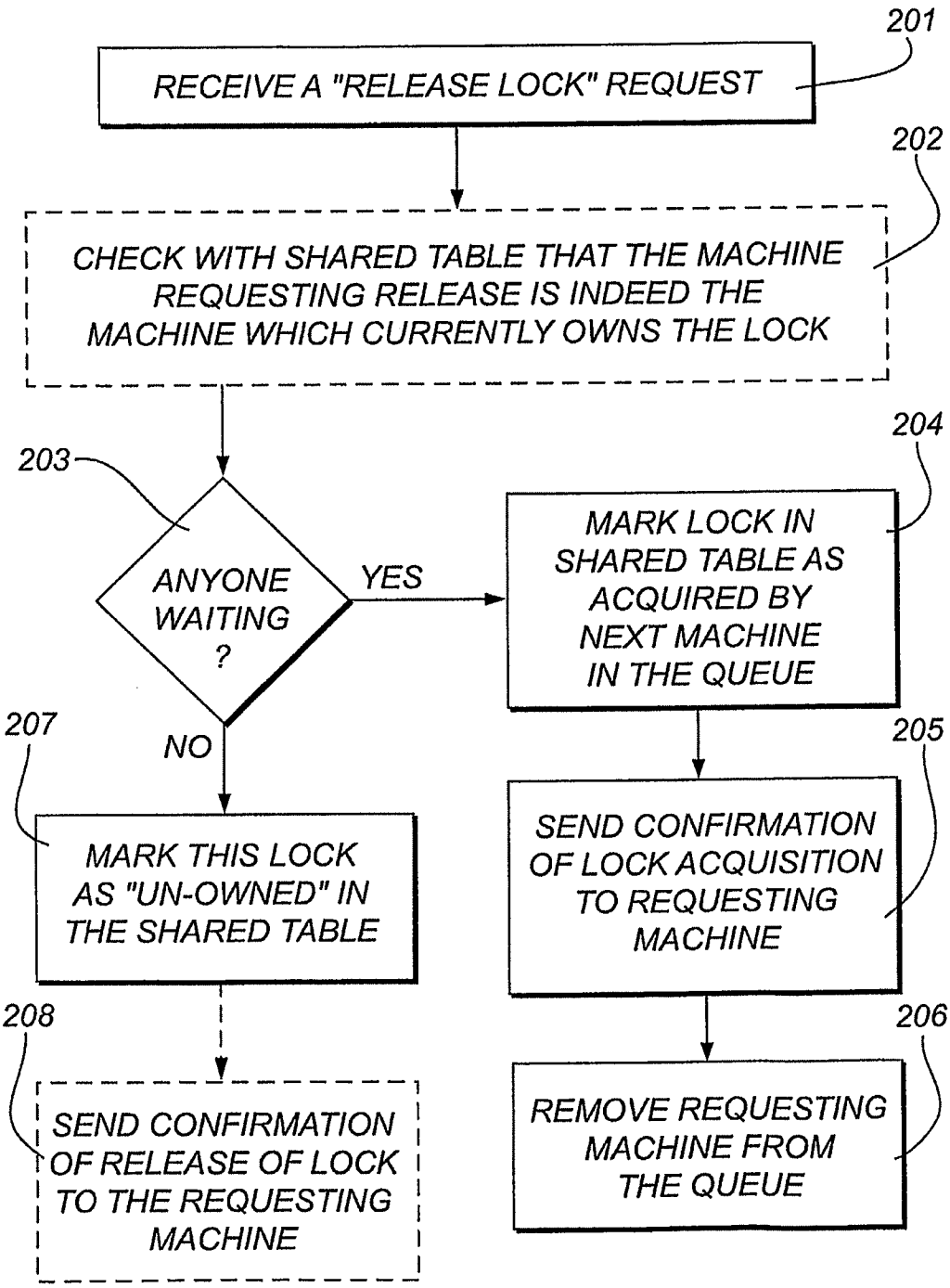
201
RECEIVE A "RELEASE LOCK" REQUEST
202
CHECK WITH SHARED TABLE THAT THE MACHINE REQUESTING RELEASE IS INDEED THE MACHINE WHICH CURRENTLY OWNS THE LOCK
203
ANYONE WAITING ?
YES
NO
204
MARK LOCK IN SHARED TABLE AS ACQUIRED BY NEXT MACHINE IN THE QUEUE
205
SEND CONFIRMATION OF LOCK ACQUISITION TO REQUESTING MACHINE
206
REMOVE REQUESTING MACHINE FROM THE QUEUE
207
MARK THIS LOCK AS "UN-OWNED" IN THE SHARED TABLE
208
SEND CONFIRMATION OF RELEASE OF LOCK TO THE REQUESTING MACHINE

FIG. 30

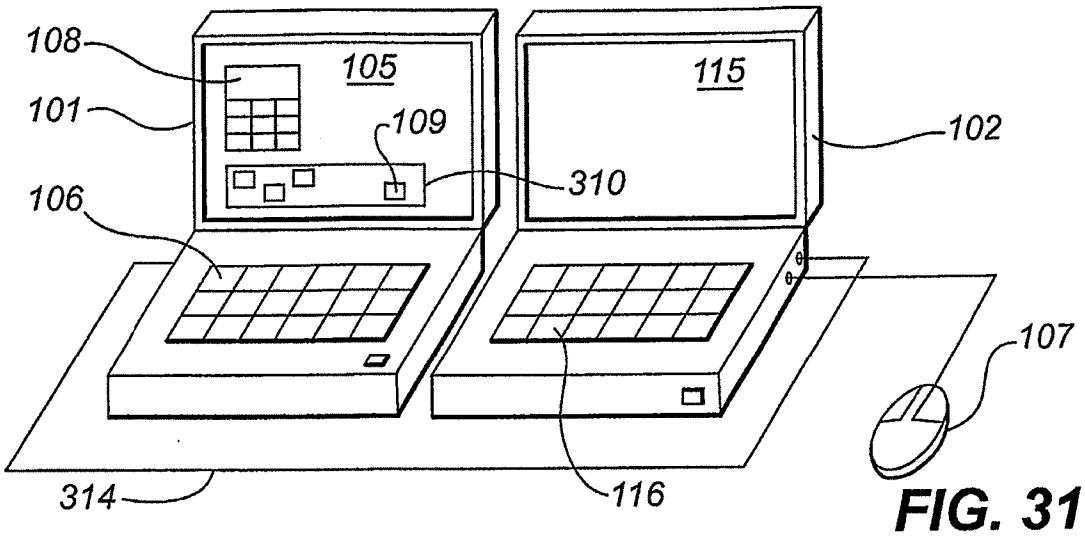

FIG. 31

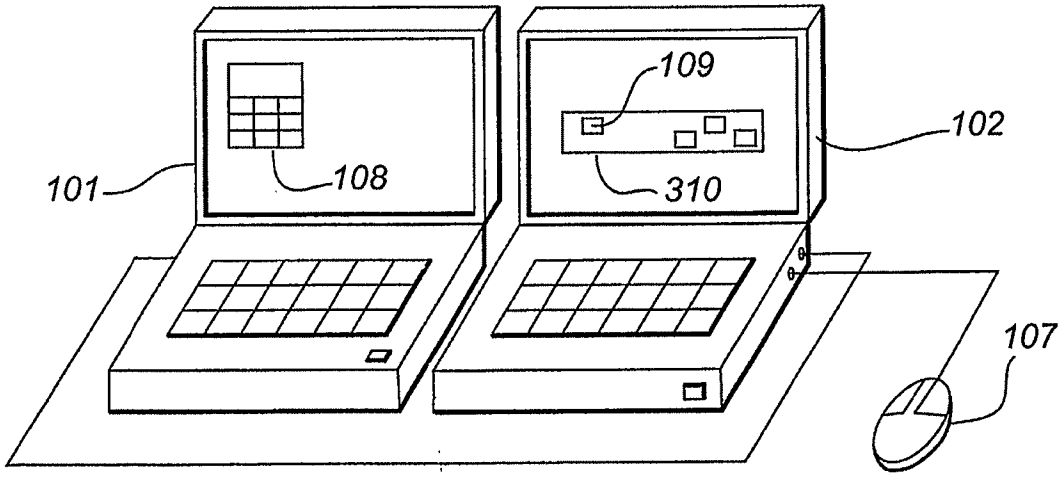

FIG. 32

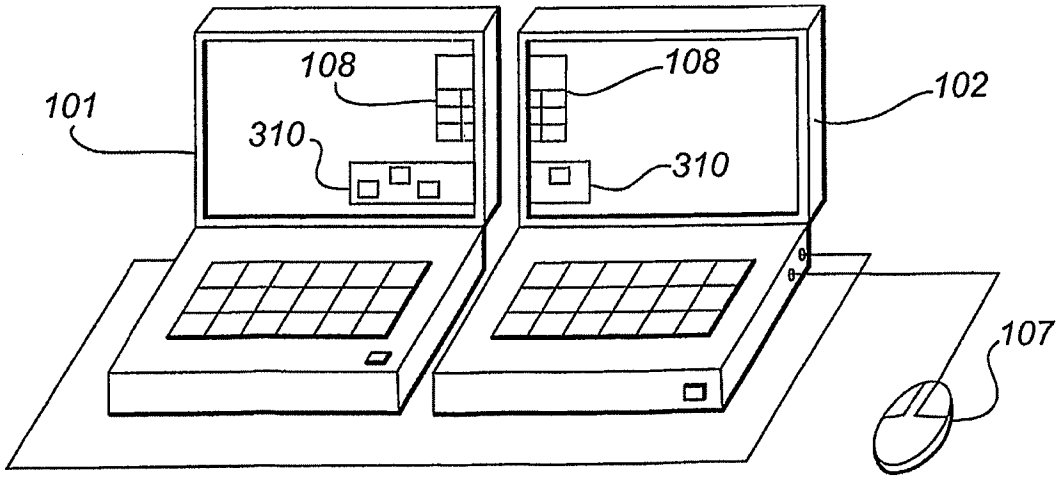

FIG. 33

INTERNATIONAL SEARCH REPORT

International application No.
PCT/AU2006/000532

A. CLASSIFICATION OF SUBJECT MATTER

Int. Cl.

G06F 15/16 (2006.01)

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
DWPI, USPTO Web Patent Database, PCT Gazette, IEEE, internet "distributed run time etc"

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
P, X	WO 2005/103924 A1 (WARATEK PTY LIMITED) 3 November 2005 Whole document.	1-139
P, X	WO 2005/103925 A1 (WARATEK PTY LIMITED) 3 November 2005 Whole document.	1-139
P, X	WO 2005/103926 A1 (WARATEK PTY LIMITED) 3 November 2005 Whole document.	1-139
P, X	WO 2005/103927 A1 (WARATEK PTY LIMITED) 3 November 2005 Whole document.	1-139
P, X	WO 2005/103928 A1 (WARATEK PTY LIMITED) 3 November 2005 Whole document.	1-139
P,X	US 2005/0240737 A1 (HOLT) 27 October 2005 Whole document.	1-139

[X] Further documents are listed in the continuation of Box C [X] See patent family annex

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
03 July 2006	10 JUL 2006
Name and mailing address of the ISA/AU AUSTRALIAN PATENT OFFICE PO BOX 200, WODEN ACT 2606, AUSTRALIA E-mail address: pct@ipaustralia.gov.au Facsimile No. (02) 6285 3929	Authorized officer **P. THONG** Telephone No : (02) 6283 2128

INTERNATIONAL SEARCH REPORT

International application No.
PCT/AU2006/000532

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	Anderson Faustino da Silva, Marcelo Lobosco, Claudio Luis de Amorim, "An Evaluation of cJava System Architecture," sbac-pad, p. 91, 5th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'03), 2003. Whole document. See URL: <http://www.cos.ufrj.br/~amorim/publications-2001-06/cJAVA-IEEE-SBAC-2003.pdf>.	29,31
X	Yariv Aridor, Michael Factor and Avi Teperman, "cJVM: A Single System Image of a JVM on a Cluster", International Conference on Parallel Processing, pages 4-11, 1999. Whole document. See URL: <http://citeseer.ist.psu.edu/cache/papers/cs/11880/http:zSzzSzwww.haifa.il.ibm.comzSzprojectszSzsystechzSzpaperszSzicpp99.pdf/aridor99cjvm.pdf>. Documents also form a single source of disclosure.	29,31
X	M. Carlsson Gothe, D. Wengelin and L. Asplund, "The Distributed Ada Run-time System DARTS", 1991. Whole document. See URL: <http://citeseer.ifi.unizh.ch/cache/papers/cs/27833/http:zSzzSzwww.cs.ubc.cazSzlocalzSzreadingzSzproceedingszSzspe91-95zSzspezSz.zSzvol21zSzissue11zSzspe064mg.pdf/gothe91distributed.pdf>.	138
X	S. H. Russ, J. Robinson, B. K. Flachs, B. Heckel, "The Hector Distributed Run-Time Environment", IEEE Transactions on Parallel and Distributed Systems, volume 9, Number 11, 1998. Whole document. See URL: <http://citeseer.ist.psu.edu/cache/papers/cs/2669/http:zSzzSzwww.erc.msstate.eduzSzthrustszSzcpszSzhectorzSzmainzSzTPDS.pdf/russ98hector.pdf>. Documents also form a single source of disclosure.	138
X	J. K. Bennett, J. B. Carter, W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence", Proc. of the Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'90), pages 168-177, year 1990. Whole document. See URL: <http://citeseer.ist.psu.edu/cache/papers/cs/1056/http:zSzzSzwww-brazos.rice.eduzSzjkbzSzpaperszSzppopp90.pdf/bennett90munin.pdf>.	1-4, 10-12 29-33, 41-44 62-65, 80-82 90-92, 133-138
X	S. Dwarkadas, P. Keleher, A. L. Cox, W. Zwaenepoel, "Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology", Proc. of the 20th Annual Int'l Symp. on Computer Architecture (ISCA'93), pages 144-155, year 1993. Whole document. See URL: <http://citeseer.ist.psu.edu/cache/papers/cs/4091/ftp:zSzzSzftp.cs.rice.eduzSzpubliczSzTreadMarkszSzpaperszSzsigarch93.pdf/dwarkadas93evaluation.pdf>. Documents also form a single source of disclosure.	1-4, 10-12 29-33, 41-44 62-65, 80-82 90-92, 133-138

INTERNATIONAL SEARCH REPORT

International application No.
PCT/AU2006/000532

Box No. II Observations where certain claims were found unsearchable (Continuation of item 2 of first sheet)

This international search report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:

1. ☐ Claims Nos.:
because they relate to subject matter not required to be searched by this Authority, namely:

2. ☐ Claims Nos.:
because they relate to parts of the international application that do not comply with the prescribed requirements to such an extent that no meaningful international search can be carried out, specifically:

3. ☐ Claims Nos.:
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a)

Box No. III Observations where unity of invention is lacking (Continuation of item 3 of first sheet)

This International Searching Authority found multiple inventions in this international application, as follows:

This International Application does not comply with the requirements of unity of invention because it does not relate to one invention or to a group of inventions so linked as to form a single general inventive concept. Where different claims have different special technical features, they define different inventions:

- Independent claims 1, 5, 8, 10, 13, 29, 41, 51, 62, 80, 88, 90, 93, 106, 116, 124, 133, 138 (and their dependent claims) relate to a multiple computer system/arrangement that suggests each computer in the arrangement operates a different portion of an application program ie a computer is provided with a different portion, such arrangement comprising a first special technical feature.
- Independent claims 22, 34, 45, 55, 66, 73, 76, 78, 83, 101, 119, 139 (and their dependent claims) relate to a multiple computer system/arrangement that suggests that an application program (rather than its portions) is present and executing on each computer ie the application is loaded onto each computer, such arrangement comprising a second special technical feature.

Since these groups of claims do not share any of the special technical features identified, a technical relationship between the inventions does not exist. Accordingly the claims do not relate to one invention or to a single inventive concept, a priori.

1. ☐ As all required additional search fees were timely paid by the applicant, this international search report covers all searchable claims.
2. ☒ As all searchable claims could be searched without effort justifying additional fees, this Authority did not invite payment of additional fees.
3. ☐ As only some of the required additional search fees were timely paid by the applicant, this international search report covers only those claims for which fees were paid, specifically claims Nos.:
4. ☐ No required additional search fees were timely paid by the applicant. Consequently, this international search report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:

Remark on Protest

☐ The additional search fees were accompanied by the applicant's protest and, where applicable, the payment of a protest fee.

☐ The additional search fees were accompanied by the applicant's protest but the applicable protest fee was not paid within the time limit specified in the invitation.

☐ No protest accompanied the payment of additional search fees.

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/AU2006/000532

This Annex lists the known "A" publication level patent family members relating to the patent documents cited in the above-mentioned international search report. The Australian Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

Patent Document Cited in Search Report		Patent Family Member					
WO	2005/103924	WO	2005103925	WO	2005103926	WO	2005103927
		WO	2005103928				
US	2005/0240737	US	2005257219	US	2005262313	US	2005262513
		US	2006020913	US	2006095483		

Due to data integration issues this family listing may not include 10 digit Australian applications filed since May 2001.

END OF ANNEX